O'REILLY®

Foundations of Scalable Systems

Designing Distributed Architectures

Ian Gorton

Foundations of Scalable Systems

Designing Distributed Architectures

This excerpt contains Chapters 1–3. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Ian Gorton

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Foundations of Scalable Systems

by Ian Gorton

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield
Development Editor: Virginia Wilson
Production Editor: Jonathon Owen
Copyeditor: Justin Billing
Proofreader: nSight, Inc.
Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2022: First Edition

Revision History for the First Edition

2022-06-29: First Release

See *https://oreil.ly/scal-sys* for release details.

This work is part of a collaboration between O'Reilly and Cockroach Labs. See our statement of editorial independence.

978-1-098-10606-5

[LSI]

Table of Contents

Foreword from Cockroach Labs

In the early days of the app economy, scale was a challenge that only a select few really had to fret over. The term "web scale" was coined to address the early challenges companies like Amazon, Google, Facebook and Netflix faced in reaching massive volumes of users. Now, though, nearly every company sees massive concurrency and increasing demands on their applications and infrastructure. Even the smallest start-up must think about scale from day one.

Modern applications are overwhelmingly data-intensive and increasingly global in nature (think Spotify, Uber, or Square). Developers now need to balance both consistency and locality around ever more components—which themselves must be managed and scaled—to satisfy users who now have zero tolerance for latency or downtime, even under the heaviest workloads. In this context, your most successful moments (your application sees rapid growth, your content goes viral on social media) can instead quickly become your worst.

But if you plan, test, and continually optimize for scale from the start, unexpected and rapid growth can simply happen without you losing sleep or scrambling to patch or repair a failing stack.

Author Ian Gorton has been in the tech industry for three decades and has compiled his hard-learned lessons into *Foundations of Scalable Systems*. You'll explore the essential ingredients of scalable solutions, including replication, state management, load balancing, and caching. Specific chapters focus on the implications of scalability for databases, microservices, and event-based streaming systems.

As a database designed for cloud native elastic scale and resilience in the face of disasters both trivial and tremendous, CockroachDB is proud to sponsor three chapters ("Introduction to Scalable Systems," "Distributed Systems Architectures: An Introduction," and "Distributed Systems Essentials") so you can learn the fundamentals of how to build for scale right from the start.

CHAPTER 1

Introduction to Scalable Systems

The last 20 years have seen unprecedented growth in the size, complexity, and capacity of software systems. This rate of growth is hardly likely to slow in the next 20 years—what future systems will look like is close to unimaginable right now. However, one thing we can guarantee is that more and more software systems will need to be built with constant growth—more requests, more data, and more analysis—as a primary design driver.

Scalable is the term used in software engineering to describe software systems that can accommodate growth. In this chapter I'll explore what precisely is meant by the ability to scale, known (not surprisingly) as *scalability*. I'll also describe a few examples that put hard numbers on the capabilities and characteristics of contemporary applications and give a brief history of the origins of the massive systems we routinely build today. Finally, I'll describe two general principles for achieving scalability, replication and optimization, which will recur in various forms throughout the rest of this book, and examine the indelible link between scalability and other software architecture quality attributes.

What Is Scalability?

Intuitively, scalability is a pretty straightforward concept. If we ask Wikipedia for a definition, it tells us, "Scalability is the property of a system to handle a growing amount of work by adding resources to the system." We all know how we scale a highway system—we add more traffic lanes so it can handle a greater number of vehicles. Some of my favorite people know how to scale beer production—they add more capacity in terms of the number and size of brewing vessels, the number of staff to perform and manage the brewing process, and the number of kegs they can fill with fresh, tasty brews. Think of any physical system—a transit system, an airport, elevators in a building—and how we increase capacity is pretty obvious.

Unlike physical systems, software systems are somewhat amorphous. They are not something you can point at, see, touch, feel, and get a sense of how it behaves internally from external observation. A software system is a digital artifact. At its core, the stream of 1s and 0s that make up executable code and data are hard for anyone to tell apart. So, what does scalability mean in terms of a software system?

Put very simply, and without getting into definition wars, scalability defines a software system's capability to handle growth in some dimension of its operations. Examples of operational dimensions are:

- The number of simultaneous user or external (e.g., sensor) requests a system can process
- The amount of data a system can effectively process and manage
- The value that can be derived from the data a system stores through predictive analytics
- The ability to maintain a stable, consistent response time as the request load grows

For example, imagine a major supermarket chain is rapidly opening new stores and increasing the number of self-checkout kiosks in every store. This requires the core supermarket software systems to perform the following functions:

- Handle increased volume from item scanning without decreased response time. Instantaneous responses to item scans are necessary to keep customers happy.
- Process and store the greater data volumes generated from increased sales. This data is needed for inventory management, accounting, planning, and likely many other functions.
- Derive "real-time" (e.g., hourly) sales data summaries from each store, region, and country and compare to historical trends. This trend data can help highlight unusual events in regions (unexpected weather conditions, large crowds at events, etc.) and help affected stores to quickly respond.
- Evolve the stock ordering prediction subsystem to be able to correctly anticipate sales (and hence the need for stock reordering) as the number of stores and customers grow.

These dimensions are effectively the scalability requirements of the system. If, over a year, the supermarket chain opens 100 new stores and grows sales by 400 times (some of the new stores are big!), then the software system needs to scale to provide the necessary processing capacity to enable the supermarket to operate efficiently. If the systems don't scale, we could lose sales when customers become unhappy. We might hold stock that will not be sold quickly, increasing costs. We might miss opportunities to increase sales by responding to local circumstances with special offerings. All these factors reduce customer satisfaction and profits. None are good for business.

Successfully scaling is therefore crucial for our imaginary supermarket's business growth, and likewise is in fact the lifeblood of many modern internet applications. But for most business and government systems, scalability is not a primary quality requirement in the early stages of development and deployment. New features to enhance usability and utility become the drivers of our development cycles. As long as performance is adequate under normal loads, we keep adding user-facing features to enhance the system's business value. In fact, introducing some of the sophisticated distributed technologies I'll describe in this book before there is a clear requirement can actually be deleterious to a project, with the additional complexity causing development inertia.

Still, it's not uncommon for systems to evolve into a state where enhanced performance and scalability become a matter of urgency, or even survival. Attractive features and high utility breed success, which brings more requests to handle and more data to manage. This often heralds a tipping point, wherein design decisions that made sense under light loads suddenly become technical debt.[1] External trigger events often cause these tipping points: look in the March/April 2020 media for the many reports of government unemployment and supermarket online ordering sites crashing under demand caused by the coronavirus pandemic.

Increasing a system's capacity in some dimension by increasing resources is called *scaling up* or *scaling out*—I'll explore the difference between these later. In addition, unlike physical systems, it is often equally important to be able to *scale down* the capacity of a system to reduce costs.

The canonical example of this is Netflix, which has a predictable regional diurnal load that it needs to process. Simply, a lot more people are watching Netflix in any geographical region at 9 p.m. than are at 5 a.m. This enables Netflix to reduce its processing resources during times of lower load. This saves the cost of running the processing nodes that are used in the Amazon cloud, as well as societally worthy things such as reducing data center power consumption. Compare this to a highway. At night when few cars are on the road, we don't retract lanes (except to make repairs). The full road capacity is available for the few drivers to go as fast as they like. In software systems, we can expand and contract our processing capacity in a matter of seconds to meet instantaneous load. Compared to physical systems, the strategies we deploy are vastly different.

1 Neil Ernst et al., *Technical Debt in Practice: How to Find It and Fix It* (MIT Press, 2021).

There's a lot more to consider about scalability in software systems, but let's come back to these issues after examining the scale of some contemporary software systems circa 2021.

Examples of System Scale in the Early 2000s

Looking ahead in this technology game is always fraught with danger. In 2008 I wrote:

> "While petabyte datasets and gigabit data streams are today's frontiers for data-intensive applications, no doubt 10 years from now we'll fondly reminisce about problems of this scale and be worrying about the difficulties that looming exascale applications are posing."[2]

Reasonable sentiments, it is true, but exascale? That's almost commonplace in today's world. Google reported multiple exabytes of Gmail in 2014, and by now, do all Google services manage a yottabyte or more? I don't know. I'm not even sure I know what a yottabyte is! Google won't tell us about their storage, but I wouldn't bet against it. Similarly, how much data does Amazon store in the various AWS data stores for their clients? And how many requests does, say, DynamoDB process per second, collectively, for all supported client applications? Think about these things for too long and your head will explode.

A great source of information that sometimes gives insights into contemporary operational scales are the major internet companies' technical blogs. There are also websites analyzing internet traffic that are highly illustrative of traffic volumes. Let's take a couple of point-in-time examples to illustrate a few things we do know today. Bear in mind these will look almost quaint in a year or four:

- Facebook's engineering blog describes Scribe, their solution for collecting, aggregating, and delivering petabytes of log data per hour, with low latency and high throughput. Facebook's computing infrastructure comprises millions of machines, each of which generates log files that capture important events relating to system and application health. Processing these log files, for example from a web server, can give development teams insights into their application's behavior and performance, and support faultfinding. Scribe is a custom buffered queuing solution that can transport logs from servers at a rate of several terabytes per second and deliver them to downstream analysis and data warehousing systems. That, my friends, is a lot of data!
- You can see live internet traffic for numerous services at Internet Live Stats. Dig around and you'll find some staggering statistics; for example, Google handles

2 Ian Gorton et al., "Data-Intensive Computing in the 21st Century," *Computer* 41, no. 4 (April 2008): 30–32.

around 3.5 billion search requests per day, Instagram users upload about 65 million photos per day, and there are something like 1.7 billion websites. It is a fun site with lots of information. Note that the data is not real, but rather estimates based on statistical analyses of multiple data sources.

- In 2016, Google published a paper describing the characteristics of its codebase. Among the many startling facts reported is the fact that "The repository contains 86 TBs of data, including approximately two billion lines of code in nine million unique source files." Remember, this was 2016.[3]

Still, real, concrete data on the scale of the services provided by major internet sites remain shrouded in commercial-in-confidence secrecy. Luckily, we can get some deep insights into the request and data volumes handled at internet scale through the annual usage report from one tech company. Beware though, as it is from Pornhub.[4] You can browse their incredibly detailed usage statistics from 2019 here. It's a fascinating glimpse into the capabilities of massive-scale systems.

How Did We Get Here? A Brief History of System Growth

I am sure many readers will have trouble believing there was civilized life before internet searching, YouTube, and social media. In fact, the first video upload to YouTube occurred in 2005. Yep, it is hard even for me to believe. So, let's take a brief look back in time at how we arrived at the scale of today's systems. Below are some historical milestones of note:

1980s
: An age dominated by time-shared mainframes and minicomputers. PCs emerged in the early 1980s but were rarely networked. By the end of the 1980s, development labs, universities, and (increasingly) businesses had email and access to primitive internet resources.

1990–95
: Networks became more pervasive, creating an environment ripe for the creation of the World Wide Web (WWW) with HTTP/HTML technology that had been pioneered at CERN by Tim Berners-Lee during the 1980s. By 1995, the number of websites was tiny, but the seeds of the future were planted with companies like Yahoo! in 1994 and Amazon and eBay in 1995.

---

3 Rachel Potvin and Josh Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM* 59, 7 (July 2016): 78–87.

4 The report is not for the squeamish. Here's one illustrative PG-13 data point—the site had 42 billion visits in 2019! Some of the statistics will definitely make your eyes bulge.

1996–2000

The number of websites grew from around 10,000 to 10 million, a truly explosive growth period. Networking bandwidth and access also grew rapidly. Companies like Amazon, eBay, Google, and Yahoo! were pioneering many of the design principles and early versions of advanced technologies for highly scalable systems that we know and use today. Everyday businesses rushed to exploit the new opportunities that e-business offered, and this brought system scalability to prominence, as explained in the sidebar "How Scale Impacted Business Systems".

2000–2006

The number of websites grew from around 10 million to 80 million during this period, and new service and business models emerged. In 2005, YouTube was launched. 2006 saw Facebook become available to the public. In the same year, Amazon Web Services (AWS), which had low-key beginnings in 2004, relaunched with its S3 and EC2 services.

2007–today

We now live in a world with around 2 billion websites, of which about 20% are active. There are something like 4 billion internet users. Huge data centers operated by public cloud operators like AWS, Google Cloud Platform (GCP), and Microsoft Azure, along with a myriad of private data centers, for example, Twitter's operational infrastructure, are scattered around the planet. Clouds host millions of applications, with engineers provisioning and operating their computational and data storage systems using sophisticated cloud management portals. Powerful cloud services make it possible for us to build, deploy, and scale our systems literally with a few clicks of a mouse. All companies have to do is pay their cloud provider bill at the end of the month.

This is the world that this book targets. A world where our applications need to exploit the key principles for building scalable systems and leverage highly scalable infrastructure platforms. Bear in mind, in modern applications, most of the code executed is not written by your organization. It is part of the containers, databases, messaging systems, and other components that you compose into your application through API calls and build directives. This makes the selection and use of these components at least as important as the design and development of your own business logic. They are architectural decisions that are not easy to change.

How Scale Impacted Business Systems

The surge of users with internet access in the 1990s brought new online moneymaking opportunities for businesses. There was a huge rush to expose business functions (sales, services, etc.) to users through a web browser. This heralded a profound change in how we had to think about building systems.

Take, for example, a retail bank. Before providing online services, it was possible to accurately predict the loads the bank's business systems would experience. We knew how many people worked in the bank and used the internal systems, how many terminals/PCs were connected to the bank's networks, how many ATMs you had to support, and the number and nature of connections to other financial institutions. Armed with this knowledge, we could build systems that support, say, a maximum of 3,000 concurrent users, safe in the knowledge that this number could not be exceeded. Growth would also be relatively slow, and most of the time (i.e., outside business hours) the load would be a lot less than the peak. This made our software design decisions and hardware provisioning a lot easier.

Now, imagine our retail bank decides to let all customers have internet banking access and the bank has five million customers. What is the maximum load now? How will load be dispersed during a business day? When are the peak periods? What happens if we run a limited time promotion to try and sign up new customers? Suddenly, our relatively simple and constrained business systems environment is disrupted by the higher average and peak loads and unpredictability you see from internet-based user populations.

Scalability Basic Design Principles

The basic aim of scaling a system is to increase its capacity in some application-specific dimension. A common dimension is increasing the number of requests that a system can process in a given time period. This is known as the system's throughput. Let's use an analogy to explore two basic principles we have available to us for scaling our systems and increasing throughput: replication and optimization.

In 1932, one of the world's iconic wonders of engineering, the Sydney Harbour Bridge, was opened. Now, it is a fairly safe assumption that traffic volumes in 2021 are somewhat higher than in 1932. If by any chance you have driven over the bridge at peak hour in the last 30 years, then you know that its capacity is exceeded considerably every day. So how do we increase throughput on physical infrastructures such as bridges?

This issue became very prominent in Sydney in the 1980s, when it was realized that the capacity of the harbor crossing had to be increased. The solution was the rather less iconic Sydney Harbour Tunnel, which essentially follows the same route underneath the harbor. This provides four additional lanes of traffic and hence added roughly one-third more capacity to harbor crossings. In not-too-far-away Auckland, their harbor bridge also had a capacity problem as it was built in 1959 with only four lanes. In essence, they adopted the same solution as Sydney, namely, to increase capacity. But rather than build a tunnel, they ingeniously doubled the number of lanes by expanding the bridge with the hilariously named "Nippon clip-ons", which widened the bridge on each side.

These examples illustrate the first strategy we have in software systems to increase capacity. We basically replicate the software processing resources to provide more capacity to handle requests and thus increase throughput, as shown in Figure 1-1. These replicated processing resources are analogous to the traffic lanes on bridges, providing a mostly independent processing pathway for a stream of arriving requests.

Luckily, in cloud-based software systems, replication can be achieved at the click of a mouse, and we can effectively replicate our processing resources thousands of times. We have it a lot easier than bridge builders in that respect. Still, we need to take care to replicate resources in order to alleviate real bottlenecks. Adding capacity to processing paths that are not overwhelmed will add needless costs without providing scalability benefit.

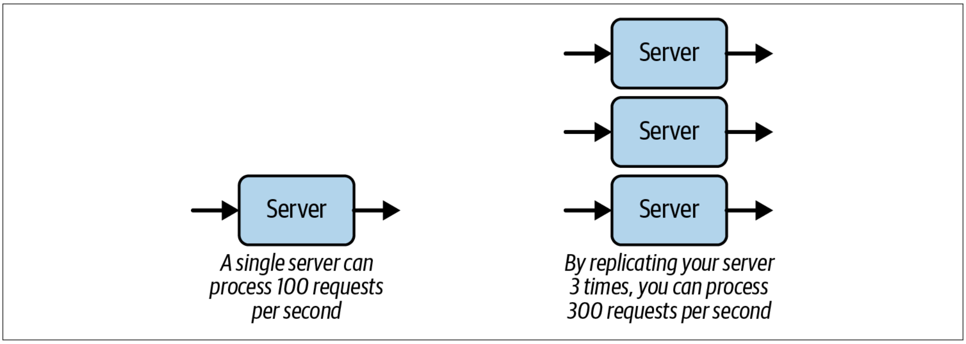

Figure 1-1. Increasing capacity through replication

The second strategy for scalability can also be illustrated with our bridge example. In Sydney, some observant person realized that in the mornings a lot more vehicles cross the bridge from north to south, and in the afternoon we see the reverse pattern. A smart solution was therefore devised—allocate more of the lanes to the high-demand direction in the morning, and sometime in the afternoon, switch this around. This effectively increased the capacity of the bridge without allocating any new resources—we *optimized* the resources we already had available.

We can follow this same approach in software to scale our systems. If we can somehow optimize our processing by using more efficient algorithms, adding extra indexes in our databases to speed up queries, or even rewriting our server in a faster programming language, we can increase our capacity without increasing our resources. The canonical example of this is Facebook's creation of (the now discontinued) HipHop for PHP, which increased the speed of Facebook's web page generation by up to six times by compiling PHP code to C++.

I'll revisit these two design principles—namely replication and optimization—throughout this book. You will see that there are many complex implications of adopting these principles, arising from the fact that we are building distributed systems. Distributed systems have properties that make building scalable systems *interesting*, which in this context has both positive and negative connotations.

Scalability and Costs

Let's take a trivial hypothetical example to examine the relationship between scalability and costs. Assume we have a web-based (e.g., web server and database) system that can service a load of 100 concurrent requests with a mean response time of 1 second. We get a business requirement to scale up this system to handle 1,000 concurrent requests with the same response time. Without making any changes, a simple load test of this system reveals the performance shown in Figure 1-2 (left). As the request load increases, we see the mean response time steadily grow to 10 seconds with the projected load. Clearly this does not satisfy our requirements in its current deployment configuration. The system doesn't scale.

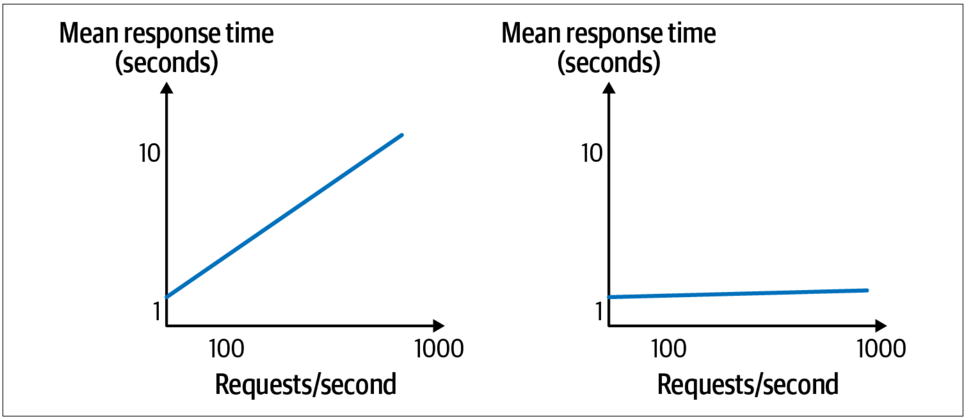

Figure 1-2. Scaling an application; non-scalable performance is represented on the left, and scalable performance on the right

Some engineering effort is needed in order to achieve the required performance. Figure 1-2 (right) shows the system's performance after this effort has been modified. It now provides the specified response time with 1,000 concurrent requests. And so, we have successfully scaled the system. Party time!

A major question looms, however. Namely, how much effort and resources were required to achieve this performance? Perhaps it was simply a case of running the web server on a more powerful (virtual) machine. Performing such reprovisioning on a cloud might take 30 minutes at most. Slightly more complex would be reconfiguring the system to run multiple instances of the web server to increase capacity. Again, this should be a simple, low-cost configuration change for the application, with no code changes needed. These would be excellent outcomes.

However, scaling a system isn't always so easy. The reasons for this are many and varied, but here are some possibilities:

- The database becomes less responsive with 1,000 requests per second, requiring an upgrade to a new machine.
- The web server generates a lot of content dynamically and this reduces response time under load. A possible solution is to alter the code to more efficiently generate the content, thus reducing processing time per request.
- The request load creates hotspots in the database when many requests try to access and update the same records simultaneously. This requires a schema redesign and subsequent reloading of the database, as well as code changes to the data access layer.
- The web server framework that was selected emphasized ease of development over scalability. The model it enforces means that the code simply cannot be scaled to meet the requested load requirements, and a complete rewrite is required. Use another framework? Use another programming language even?

There's a myriad of other potential causes, but hopefully these illustrate the increasing effort that might be required as we move from possibility (1) to possibility (4).

Now let's assume option (1), upgrading the database server, requires 15 hours of effort and a thousand dollars in extra cloud costs per month for a more powerful server. This is not prohibitively expensive. And let's assume option (4), a rewrite of the web application layer, requires 10,000 hours of development due to implementing a new language (e.g., Java instead of Ruby). Options (2) and (3) fall somewhere in between options (1) and (4). The cost of 10,000 hours of development is seriously significant. Even worse, while the development is underway, the application may be losing market share and hence money due to its inability to satisfy client requests' loads. These kinds of situations can cause systems and businesses to fail.

This simple scenario illustrates how the dimensions of resource and effort costs are inextricably tied to scalability. If a system is not designed intrinsically to scale, then the downstream costs and resources of increasing its capacity to meet requirements may be massive. For some applications, such as HealthCare.gov, these (more than $2 billion) costs are borne and the system is modified to eventually meet business needs. For others, such as Oregon's health care exchange, an inability to scale rapidly at low cost can be an expensive ($303 million, in Oregon's case) death knell.

We would never expect someone would attempt to scale up the capacity of a suburban home to become a 50-floor office building. The home doesn't have the architecture, materials, and foundations for this to be even a remote possibility without being completely demolished and rebuilt. Similarly, we shouldn't expect software systems that do not employ scalable architectures, mechanisms, and technologies to be quickly evolved to meet greater capacity needs. The foundations of scale need to be built in from the beginning, with the recognition that the components will evolve over time. By employing design and development principles that promote scalability, we can more rapidly and cheaply scale up systems to meet rapidly growing demands. I'll explain these principles in Part II of this book.

Software systems that can be scaled exponentially while costs grow linearly are known as hyperscale systems, which I define as follows: "Hyper scalable systems exhibit exponential growth in computational and storage capabilities while exhibiting linear growth rates in the costs of resources required to build, operate, support, and evolve the required software and hardware resources." You can read more about hyperscale systems in this article.

Scalability and Architecture Trade-Offs

Scalability is just one of the many quality attributes, or nonfunctional requirements, that are the lingua franca of the discipline of software architecture. One of the enduring complexities of software architecture is the necessity of quality attribute trade-offs. Basically, a design that favors one quality attribute may negatively or positively affect others. For example, we may want to write log messages when certain events occur in our services so we can do forensics and support debugging of our code. We need to be careful, however, how many events we capture, because logging introduces overheads and negatively affects performance and cost.

Experienced software architects constantly tread a fine line, crafting their designs to satisfy high-priority quality attributes, while minimizing the negative effects on other quality attributes.

Scalability is no different. When we point the spotlight at the ability of a system to scale, we have to carefully consider how our design influences other highly desirable properties such as performance, availability, security, and the oft overlooked manageability. I'll briefly discuss some of these inherent trade-offs in the following sections.

Performance

There's a simple way to think about the difference between performance and scalability. When we target performance, we attempt to satisfy some desired metrics for individual requests. This might be a mean response time of less than 2 seconds, or a worst-case performance target such as the 99th percentile response time less than 3 seconds.

Improving performance is in general a good thing for scalability. If we improve the performance of individual requests, we create more capacity in our system, which helps us with scalability as we can use the unused capacity to process more requests.

However, it's not always that simple. We may reduce response times in a number of ways. We might carefully optimize our code by, for example, removing unnecessary object copying, using a faster JSON serialization library, or even completely rewriting code in a faster programming language. These approaches optimize performance without increasing resource usage.

An alternative approach might be to optimize individual requests by keeping commonly accessed state in memory rather than writing to the database on each request. Eliminating a database access nearly always speeds things up. However, if our system maintains large amounts of state in memory for prolonged periods, we may (and in a heavily loaded system, will) have to carefully manage the number of requests our system can handle. This will likely reduce scalability as our optimization approach for individual requests uses more resources (in this case, memory) than the original solution, and thus reduces system capacity.

We'll see this tension between performance and scalability reappear throughout this book. In fact, it's sometimes judicious to make individual requests slightly slower so we can utilize additional system capacity. A great example of this is described when I discuss load balancing in the next chapter.

Availability

Availability and scalability are in general highly compatible partners. As we scale our systems through replicating resources, we create multiple instances of services that can be used to handle requests from any users. If one of our instances fails, the others remain available. The system just suffers from reduced capacity due to a failed, unavailable resource. Similar thinking holds for replicating network links, network routers, disks, and pretty much any resource in a computing system.

Things get complicated with scalability and availability when state is involved. Think of a database. If our single database server becomes overloaded, we can replicate it and send requests to either instance. This also increases availability as we can tolerate the failure of one instance. This scheme works great if our databases are read only. But as soon as we update one instance, we somehow have to figure out how and when to update the other instance. This is where the issue of replica consistency raises its ugly head.

In fact, whenever state is replicated for scalability and availability, we have to deal with consistency. This will be a major topic when I discuss distributed databases in Part III of this book.

Security

Security is a complex, highly technical topic worthy of its own book. No one wants to use an insecure system, and systems that are hacked and compromise user data cause CTOs to resign, and in extreme cases, companies to fail.

The basic elements of a secure system are authentication, authorization, and integrity. We need to ensure data cannot be intercepted in transit over networks, and data at rest (persistent store) cannot be accessed by anyone who does not have permission to access that data. Basically, I don't want anyone seeing my credit card number as it is communicated between systems or stored in a company's database.

Hence, security is a necessary quality attribute for any internet-facing systems. The costs of building secure systems cannot be avoided, so let's briefly examine how these affect performance and scalability.

At the network level, systems routinely exploit the Transport Layer Security (TLS) protocol, which runs on top of TCP/IP (see Chapter 3). TLS provides encryption, authentication, and integrity using asymmetric cryptography. This has a performance cost for establishing a secure connection as both parties need to generate and exchange keys. TLS connection establishment also includes an exchange of certificates to verify the identity of the server (and optionally client), and the selection of an algorithm to check that the data is not tampered with in transit. Once a connection is established, in-flight data is encrypted using symmetric cryptography, which has a negligible performance penalty as modern CPUs have dedicated encryption hardware. Connection establishment usually requires two message exchanges between client and server, and is thus comparatively slow. Reusing connections as much as possible minimizes these performance overheads.

There are multiple options for protecting data at rest. Popular database engines such as SQL Server and Oracle have features such as transparent data encryption (TDE) that provides efficient file-level encryption. Finer-grain encryption mechanisms, down to field level, are increasingly required in regulated industries such as finance. Cloud providers offer various features too, ensuring data stored in cloud-based data stores is secure. The overheads of secure data at rest are simply costs that must be borne to achieve security—studies suggest the overheads are in the 5–10% range.

Another perspective on security is the CIA triad, which stands for *confidentiality*, *integrity*, and *availability*. The first two are pretty much what I have described above. Availability refers to a system's ability to operate reliably under attack from adversaries. Such attacks might be attempts to exploit a system design weakness to bring the system down. Another attack is the classic distributed denial-of-service (DDoS), in which an adversary gains control over multitudes of systems and devices and coordinates a flood of requests that effectively make a system unavailable.

In general, security and scalability are opposing forces. Security necessarily introduces performance degradation. The more layers of security a system encompasses, then a greater burden is placed on performance, and hence scalability. This eventually affects the bottom line—more powerful and expensive resources are required to achieve a system's performance and scalability requirements.

Manageability

As the systems we build become more distributed and complex in their interactions, their management and operations come to the fore. We need to pay attention to ensuring every component is operating as expected, and the performance is continuing to meet expectations.

The platforms and technologies we use to build our systems provide a multitude of standards-based and proprietary monitoring tools that can be used for these purposes. Monitoring dashboards can be used to check the ongoing health and behavior of each system component. These dashboards, built using highly customizable and open tools such as Grafana, can display system metrics and send alerts when various thresholds or events occur that need operator attention. The term used for this sophisticated monitoring capability is *observability*.

There are various APIs such as Java's MBeans, AWS CloudWatch and Python's AppMetrics that engineers can utilize to capture custom metrics for their systems—a typical example is request response times. Using these APIs, monitoring dashboards can be tailored to provide live charts and graphs that give deep insights into a system's behavior. Such insights are invaluable to ensure ongoing operations and highlight parts of the system that may need optimization or replication.

Scaling a system invariably means adding new system components—hardware and software. As the number of components grows, we have more moving parts to monitor and manage. This is never effort-free. It adds complexity to the operations of the system and costs in terms of monitoring code that requires developing and observability platform evolution.

The only way to control the costs and complexity of manageability as we scale is through automation. This is where the world of DevOps enters the scene. *DevOps* is a set of practices and tooling that combine software development and system operations. DevOps reduces the development lifecycle for new features and automates ongoing test, deployment, management, upgrade, and monitoring of the system. It's an integral part of any successful scalable system.

Summary and Further Reading

The ability to scale an application quickly and cost-effectively should be a defining quality of the software architecture of contemporary internet-facing applications. We have two basic ways to achieve scalability, namely increasing system capacity, typically through replication, and performance optimization of system components.

Like any software architecture quality attribute, scalability cannot be achieved in isolation. It inevitably involves complex trade-offs that need to be tuned to an application's requirements. I'll be discussing these fundamental trade-offs throughout the remainder of this book, starting in the next chapter when I describe concrete architecture approaches to achieve scalability.

CHAPTER 2

Distributed Systems Architectures: An Introduction

In this chapter, I'll broadly cover some of the fundamental approaches to scaling a software system. You can regard this as a 30,000-foot view of the content that is covered in Part II, Part III, and Part IV of this book. I'll take you on a tour of the main architectural approaches used for scaling a system, and give pointers to later chapters where these issues are dealt with in depth. You can think of this as an overview of why we need these architectural tactics, with the remainder of the book explaining the how.

The type of systems this book is oriented toward are the internet-facing systems we all utilize every day. I'll let you name your favorite. These systems accept requests from users through web and mobile interfaces, store and retrieve data based on user requests or events (e.g., a GPS-based system), and have some *intelligent* features such as providing recommendations or notifications based on previous user interactions.

I'll start with a simple system design and show how it can be scaled. In the process, I'll introduce several concepts that will be covered in much more detail later in this book. This chapter just gives a broad overview of these concepts and how they aid in scalability—truly a whirlwind tour!

Basic System Architecture

Virtually all massive-scale systems start off small and grow due to their success. It's common, and sensible, to start with a development framework such as Ruby on Rails, Django, or equivalent, which promotes rapid development to get a system quickly up and running. A typical very simple software architecture for "starter" systems, which closely resembles what you get with rapid development frameworks, is shown in Figure 2-1. This comprises a client tier, application service tier, and a database tier. If you use Rails or equivalent, you also get a framework which hardwires a model–view–controller (MVC) pattern for web application processing and an object–relational mapper (ORM) that generates SQL queries.

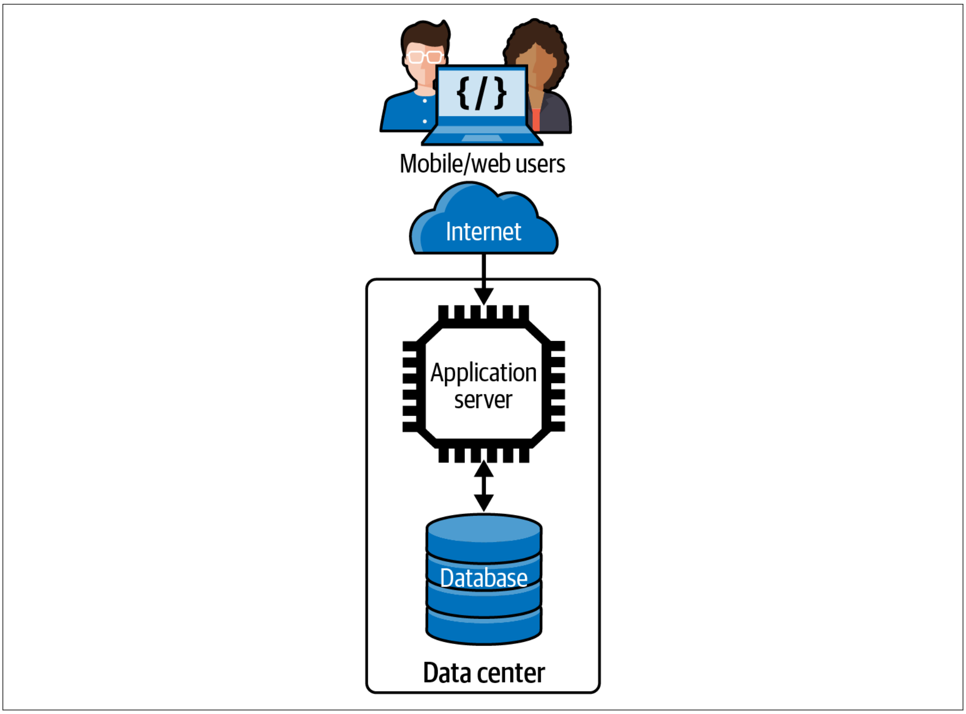

Figure 2-1. Basic multitier distributed systems architecture

With this architecture, users submit requests to the application from their mobile app or web browser. The magic of internet networking (see Chapter 3) delivers these requests to the application service which is running on a machine hosted in some corporate or commercial cloud data center. Communications uses a standard application-level network protocol, typically HTTP.

The application service runs code supporting an API that clients use to send HTTP requests. Upon receipt of a request, the service executes the code associated with the requested API. In the process, it may read from or write to a database or some other external system, depending on the semantics of the API. When the request is complete, the service sends the results to the client to display in their app or browser.

Many, if not most systems conceptually look exactly like this. The application service code exploits a server execution environment that enables multiple requests from multiple users to be processed simultaneously. There's a myriad of these application server technologies—for example, Java EE and the Spring Framework for Java, Flask for Python—that are widely used in this scenario.

This approach leads to what is generally known as a monolithic architecture.[1] Monoliths tend to grow in complexity as the application becomes more feature-rich. All API handlers are built into the same server code body. This eventually makes it hard to modify and test rapidly, and the execution footprint can become extremely heavyweight as all the API implementations run in the same application service.

Still, if request loads stay relatively low, this application architecture can suffice. The service has the capacity to process requests with consistently low latency. But if request loads keep growing, this means latencies will increase as the service has insufficient CPU/memory capacity for the concurrent request volume and therefore requests will take longer to process. In these circumstances, our single server is overloaded and has become a bottleneck.

In this case, the first strategy for scaling is usually to "scale up" the application service hardware. For example, if your application is running on AWS, you might upgrade your server from a modest t3.xlarge instance with four (virtual) CPUs and 16 GB of memory to a t3.2xlarge instance, which doubles the number of CPUs and memory available for the application.[2]

Scaling up is simple. It gets many real-world applications a long way to supporting larger workloads. It obviously costs more money for hardware, but that's scaling for you.

It's inevitable, however, that for many applications the load will grow to a level which will swamp a single server node, no matter how many CPUs and how much memory you have. That's when you need a new strategy—namely, scaling out, or horizontal scaling, which I touched on in Chapter 1.

Scale Out

Scaling out relies on the ability to replicate a service in the architecture and run multiple copies on multiple server nodes. Requests from clients are distributed across the replicas so that in theory, if we have N replicas and R requests, each server node processes R/N requests. This simple strategy increases an application's capacity and hence scalability.

To successfully scale out an application, you need two fundamental elements in your design. As illustrated in Figure 2-2, these are:

1 Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach* (O'Reilly Media, 2020).

2 See Amazon EC2 Instance Types for a description of AWS instances.

A load balancer

All user requests are sent to a load balancer, which chooses a service replica target to process the request. Various strategies exist for choosing a target service, all with the core aim of keeping each resource equally busy. The load balancer also relays the responses from the service back to the client. Most load balancers belong to a class of internet components known as reverse proxies. These control access to server resources for client requests. As an intermediary, reverse proxies add an extra network hop for a request; they need to be extremely low latency to minimize the overheads they introduce. There are many off-the-shelf load balancing solutions as well as cloud provider–specific ones, and I'll cover the general characteristics of these in much more detail in Chapter 5.

Stateless services

For load balancing to be effective and share requests evenly, the load balancer must be free to send consecutive requests from the same client to different service instances for processing. This means the API implementations in the services must retain no knowledge, or state, associated with an individual client's session. When a user accesses an application, a user session is created by the service and a unique session is managed internally to identify the sequence of user interactions and track session state. A classic example of session state is a shopping cart. To use a load balancer effectively, the data representing the current contents of a user's cart must be stored somewhere—typically a data store—such that any service replica can access this state when it receives a request as part of a user session. In Figure 2-2, this is labeled as a "Session store."

Scaling out is attractive as, in theory, you can keep adding new (virtual) hardware and services to handle increased request loads and keep request latencies consistent and low. As soon as you see latencies rising, you deploy another server instance. This requires no code changes with stateless services and is relatively cheap as a result—you just pay for the hardware you deploy.

Scaling out has another highly attractive feature. If one of the services fails, the requests it is processing will be lost. But as the failed service manages no session state, these requests can be simply reissued by the client and sent to another service instance for processing. This means the application is resilient to failures in the service software and hardware, thus enhancing the application's availability.

Unfortunately, as with any engineering solution, simple scaling out has limits. As you add new service instances, the request processing capacity grows, potentially infinitely. At some stage, however, reality will bite and the capability of your single database to provide low-latency query responses will diminish. Slow queries will mean longer response times for clients. If requests keep arriving faster than they are being processed, some system components will become overloaded and fail due to resource exhaustion, and clients will see exceptions and request timeouts. Essentially, your database becomes a bottleneck that you must engineer away in order to scale your application further.

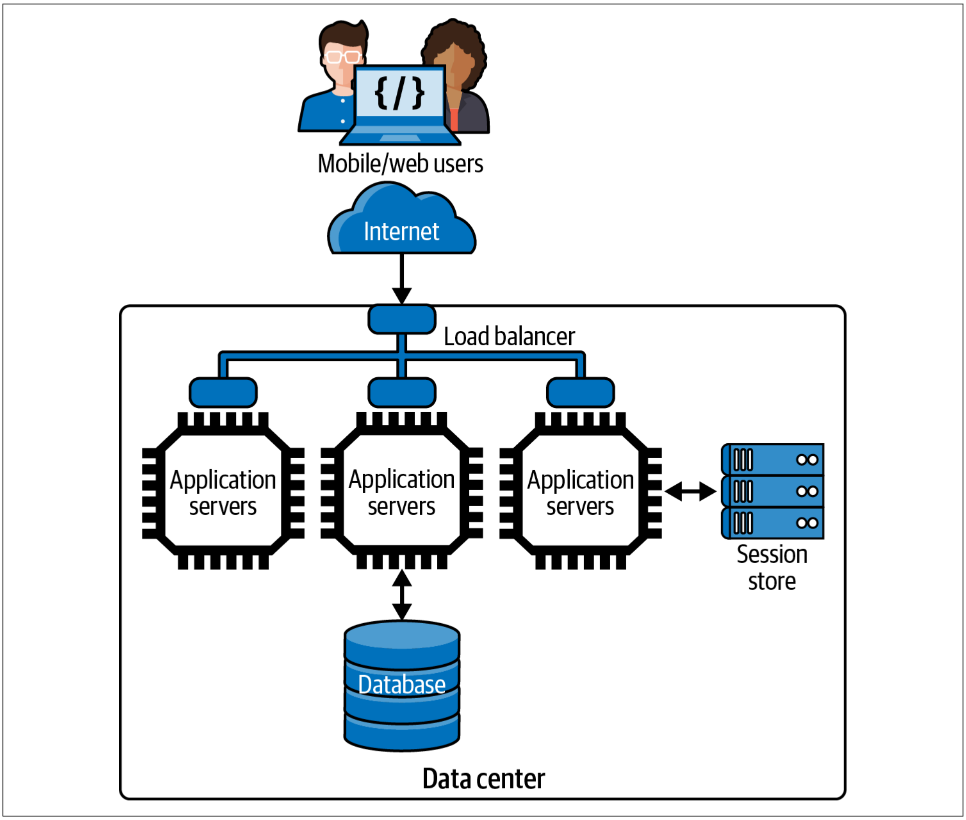

Figure 2-2. Scale-out architecture

Scaling the Database with Caching

Scaling up by increasing the number of CPUs, memory, and disks in a database server can go a long way to scaling a system. For example, at the time of writing, GCP can provision a SQL database on a db-n1-highmem-96 node, which has 96 virtual CPUs (vCPUs), 624 GB of memory, 30 TB of disk, and can support 4,000 connections. This will cost somewhere between $6K and $16K per year, which sounds like a good deal to me! Scaling up is a common database scalability strategy.

Large databases need constant care and attention from highly skilled database administrators to keep them tuned and running fast. There's a lot of wizardry in this job—e.g., query tuning, disk partitioning, indexing, on-node caching, and so on—and hence database administrators are valuable people you want to be very nice to. They can make your application services highly responsive.

In conjunction with scaling up, a highly effective approach is querying the database as infrequently as possible from your services. This can be achieved by employing *distributed caching* in the scaled-out service tier. Caching stores recently retrieved and commonly accessed database results in memory so they can be quickly retrieved without placing a burden on the database. For example, the weather forecast for the next hour won't change, but may be queried by hundreds or thousands of clients. You can use a cache to store the forecast once it is issued. All client requests will read from the cache until the forecast expires.

For data that is frequently read and changes rarely, your processing logic can be modified to first check a distributed cache, such as a Redis or memcached store. These cache technologies are essentially distributed key-value stores with very simple APIs. This scheme is illustrated in Figure 2-3. Note that the session store from Figure 2-2 has disappeared. This is because you can use a general-purpose distributed cache to store session identifiers along with application data.

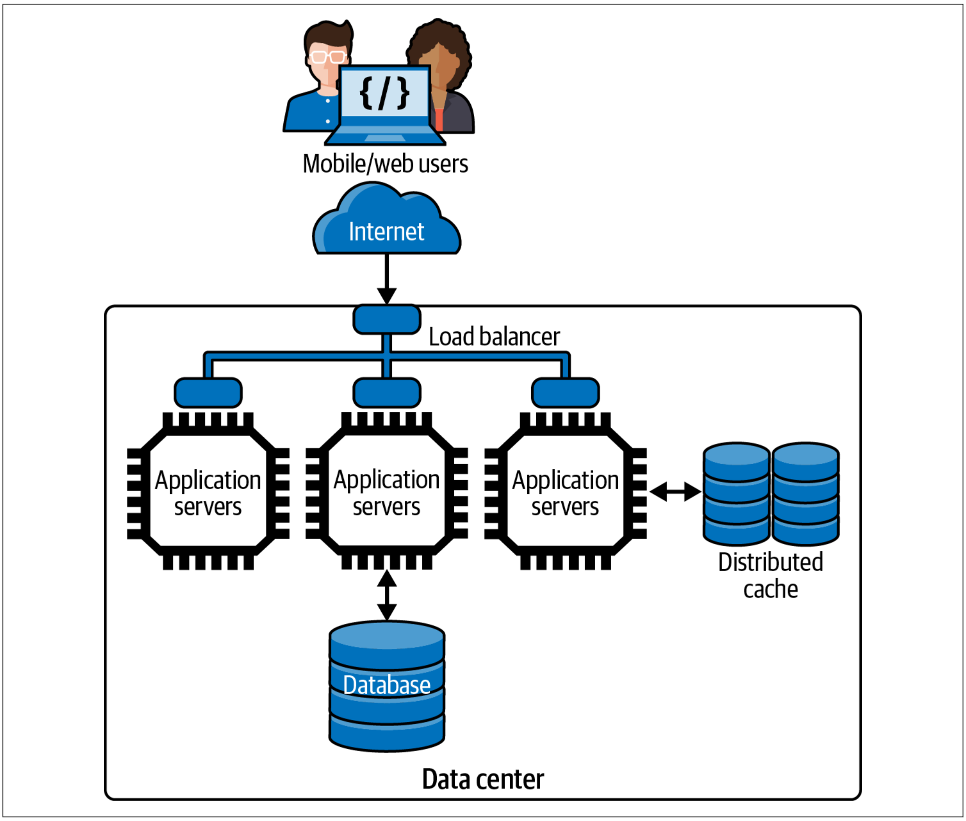

Figure 2-3. Introducing distributed caching

Accessing the cache requires a remote call from your service. If the data you need is in the cache, on a fast network you can expect submillisecond cache reads. This is far less expensive than querying the shared database instance, and also doesn't require a query to contend for typically scarce database connections.

Introducing a caching layer also requires your processing logic to be modified to check for cached data. If what you want is not in the cache, your code must still query the database and load the results into the cache as well as return it to the caller. You also need to decide when to remove, or invalidate, cached results—your course of action depends on the nature of your data (e.g., weather forecasts expire naturally) and your application's tolerance to serving out-of-date—also known as *stale*—results to clients.

A well-designed caching scheme can be invaluable in scaling a system. Caching works great for data that rarely changes and is accessed frequently, such as inventory catalogs, event information, and contact data. If you can handle a large percentage, say, 80% or more, of read requests from your cache, then you effectively buy extra capacity at your databases as they never see a large proportion of requests.

Still, many systems need to rapidly access terabytes and larger data stores that make a single database effectively prohibitive. In these systems, a distributed database is needed.

Distributing the Database

There are more distributed database technologies around in 2022 than you probably want to imagine. It's a complex area, and one I'll cover extensively in Part III. In very general terms, there are two major categories:

Distributed SQL stores
: These enable organizations to scale out their SQL database relatively seamlessly by storing the data across multiple disks that are queried by multiple database engine replicas. These multiple engines logically appear to the application as a single database, hence minimizing code changes. There is also a class of "born distributed" SQL databases that are commonly known as NewSQL stores that fit in this category.

Distributed so-called "NoSQL" stores (from a whole array of vendors)
: These products use a variety of data models and query languages to distribute data across multiple nodes running the database engine, each with their own locally attached storage. Again, the location of the data is transparent to the application, and typically controlled by the design of the data model using hashing functions on database keys. Leading products in this category are Cassandra, MongoDB, and Neo4j.

Figure 2-4 shows how our architecture incorporates a distributed database. As the data volumes grow, a distributed database can increase the number of storage nodes. As nodes are added (or removed), the data managed across all nodes is rebalanced to attempt to ensure the processing and storage capacity of each node is equally utilized.

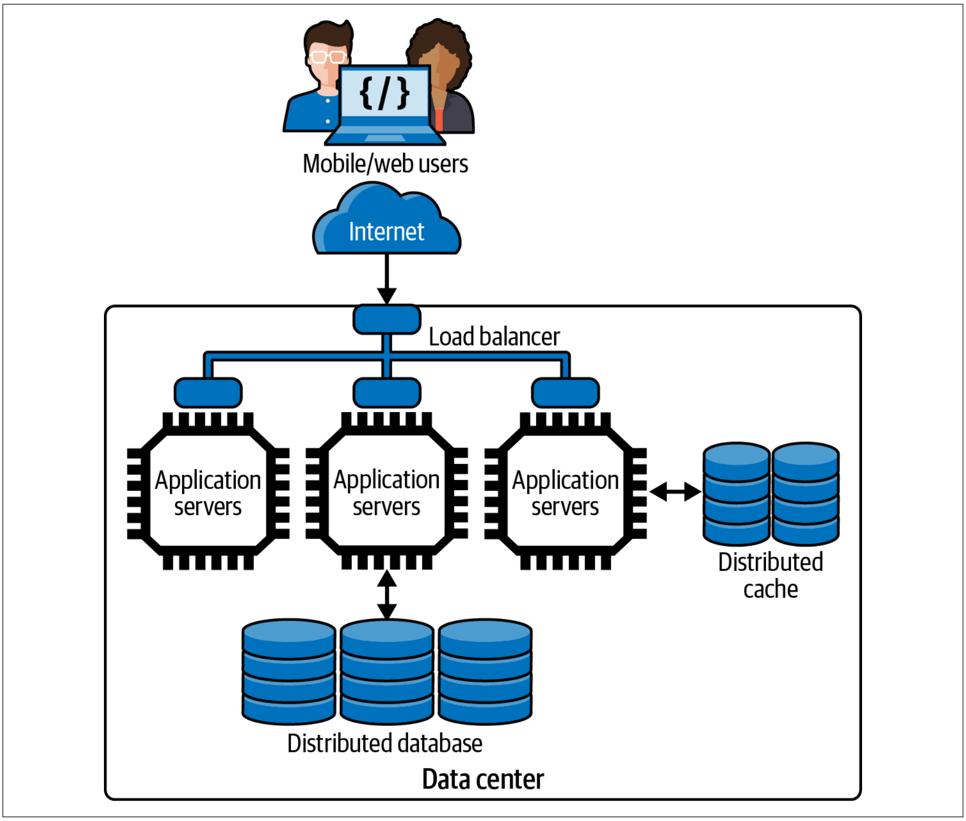

Figure 2-4. Scaling the data tier using a distributed database

Distributed databases also promote availability. They support replicating each data storage node so if one fails or cannot be accessed due to network problems, another copy of the data is available. The models utilized for replication and the trade-offs these require (spoiler alert: consistency) are covered in later chapters.

If you are utilizing a major cloud provider, there are also two deployment choices for your data tier. You can deploy your own virtual resources and build, configure, and administer your own distributed database servers. Alternatively, you can utilize cloud-hosted databases. The latter simplifies the administrative effort associated with managing, monitoring, and scaling the database, as many of these tasks essentially become the responsibility of the cloud provider you choose. As usual, the no free lunch principle applies. You always pay, whichever approach you choose.

Multiple Processing Tiers

Any realistic system that you need to scale will have many different services that interact to process a request. For example, accessing a web page on Amazon.com can require in excess of 100 different services being called before a response is returned to the user.[3]

The beauty of the stateless, load-balanced, cached architecture I am elaborating in this chapter is that it's possible to extend the core design principles and build a multitiered application. In fulfilling a request, a service can call one or more dependent services, which in turn are replicated and load-balanced. A simple example is shown in Figure 2-5. There are many nuances in how the services interact, and how applications ensure rapid responses from dependent services. Again, I'll cover these in detail in later chapters.

3 Werner Vogels, "Modern Applications at AWS," All Things Distributed, 28 Aug. 2019, *https://oreil.ly/FXOep*.

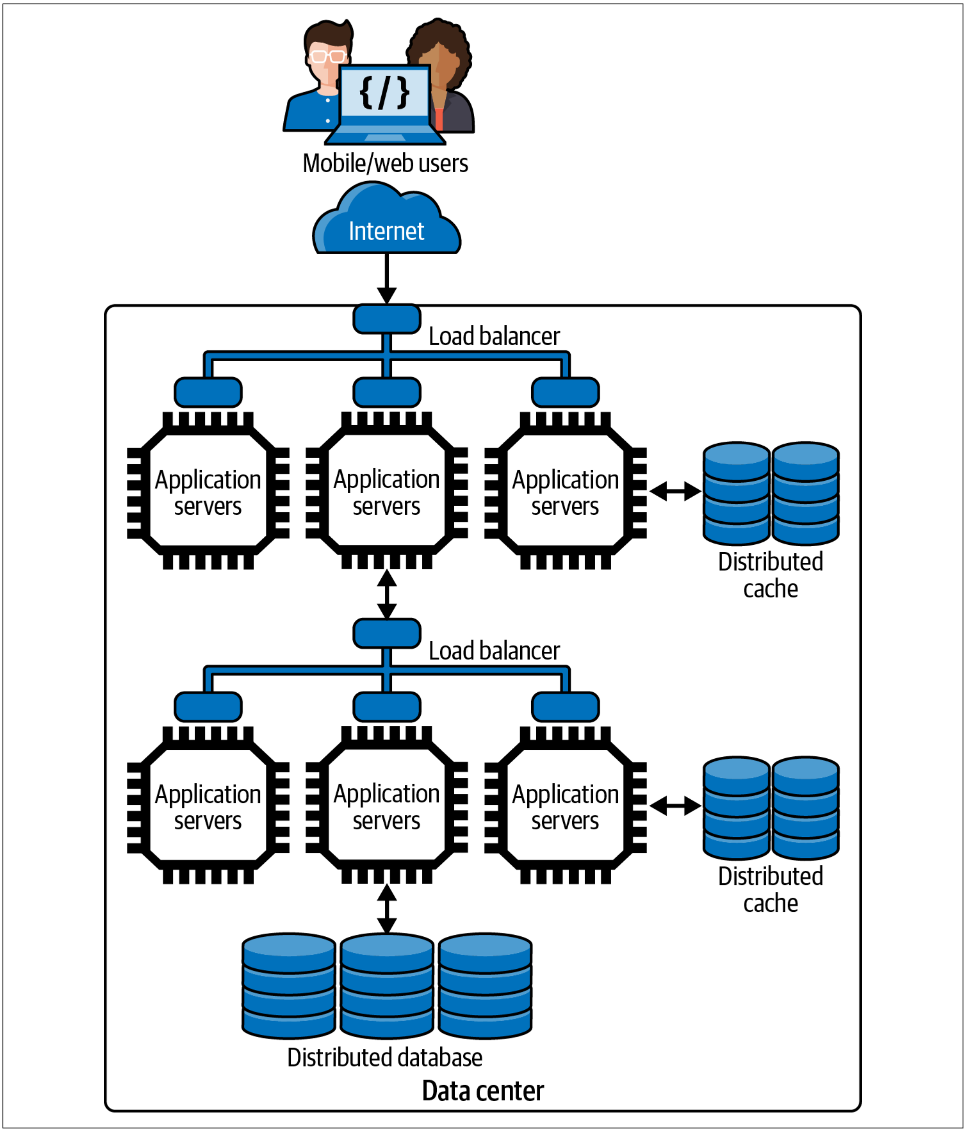

Figure 2-5. Scaling processing capacity with multiple tiers

This design also promotes having different, load-balanced services at each tier in the architecture. For example, Figure 2-6 illustrates two replicated internet-facing services that both utilized a core service that provides database access. Each service is load balanced and employs caching to provide high performance and availability. This design is often used to provide a service for web clients and a service for mobile clients, each of which can be scaled independently based on the load they experience. It's commonly called the Backend for Frontend (BFF) pattern.[4]

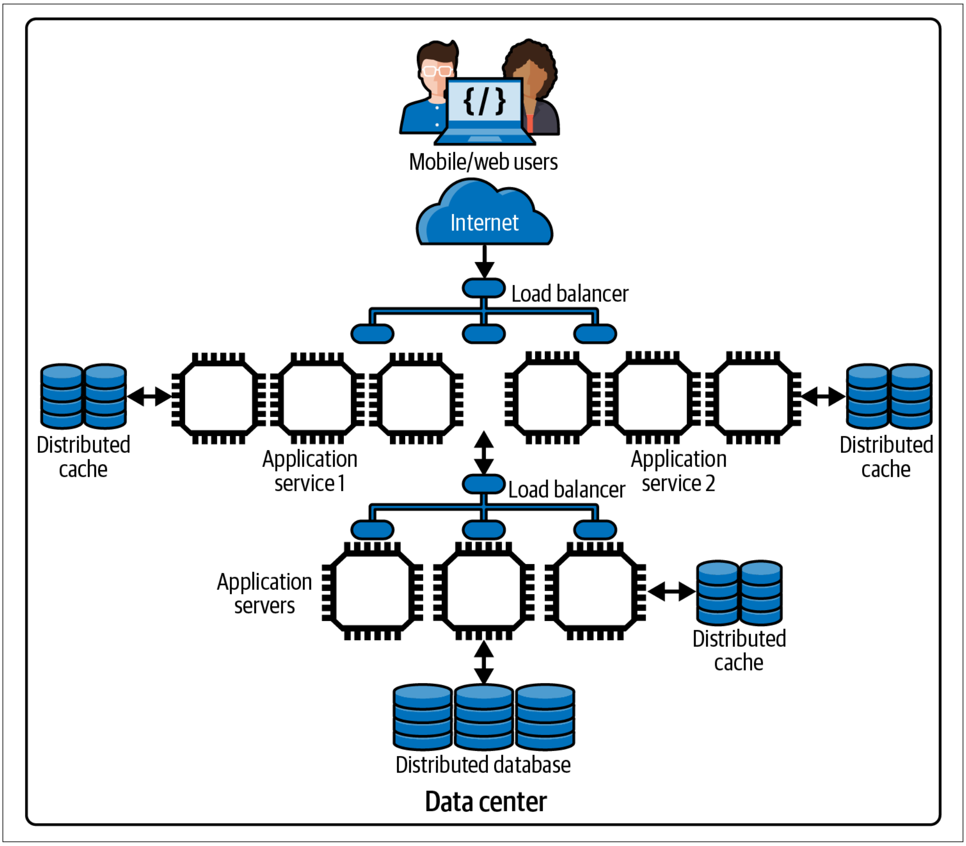

Figure 2-6. Scalable architecture with multiple services

In addition, by breaking the application into multiple independent services, you can scale each based on the service demand. If, for example, you see an increasing volume of requests from mobile users and decreasing volumes from web users, it's possible to provision different numbers of instances for each service to satisfy demand. This is a major advantage of refactoring monolithic applications into multiple independent services, which can be separately built, tested, deployed, and scaled. I'll explore some of the major issues in designing systems based on such services, known as microservices, in Chapter 9.

4 Sam Newman, "Pattern: Backends For Frontends," Sam Newman & Associates, November 18, 2015. *https://oreil.ly/1KR1z*.

Increasing Responsiveness

Most client application requests expect a response. A user might want to see all auction items for a given product category or see the real estate that is available for sale in a given location. In these examples, the client sends a request and waits until a response is received. This time interval between sending the request and receiving the result is the response time of the request. You can decrease response times by using caching and precalculated responses, but many requests will still result in database accesses.

A similar scenario exists for requests that update data in an application. If a user updates their delivery address immediately prior to placing an order, the new delivery address must be persisted so that the user can confirm the address before they hit the "purchase" button. The response time in this case includes the time for the database write, which is confirmed by the response the user receives.

Some update requests, however, can be successfully responded to without fully persisting the data in a database. For example, the skiers and snowboarders among you will be familiar with lift ticket scanning systems that check you have a valid pass to ride the lifts that day. They also record which lifts you take, the time you get on, and so on. Nerdy skiers/snowboarders can then use the resort's mobile app to see how many lifts they ride in a day.

As a person waits to get on a lift, a scanner device validates the pass using an RFID (radio-frequency identification) chip reader. The information about the rider, lift, and time are then sent over the internet to a data capture service operated by the ski resort. The lift rider doesn't have to wait for this to occur, as the response time could slow down the lift-loading process. There's also no expectation from the lift rider that they can instantly use their app to ensure this data has been captured. They just get on the lift, talk smack with their friends, and plan their next run.

Service implementations can exploit this type of scenario to improve responsiveness. The data about the event is sent to the service, which acknowledges receipt and concurrently stores the data in a remote queue for subsequent writing to the database. Distributed queueing platforms can be used to reliably send data from one service to another, typically but not always in a first-in, first-out (FIFO) manner.

Writing a message to a queue is typically much faster than writing to a database, and this enables the request to be successfully acknowledged much more quickly. Another service is deployed to read messages from the queue and write the data to the database. When a skier checks their lift rides—maybe three hours or three days later—the data has been persisted successfully in the database.

The basic architecture to implement this approach is illustrated in Figure 2-7.

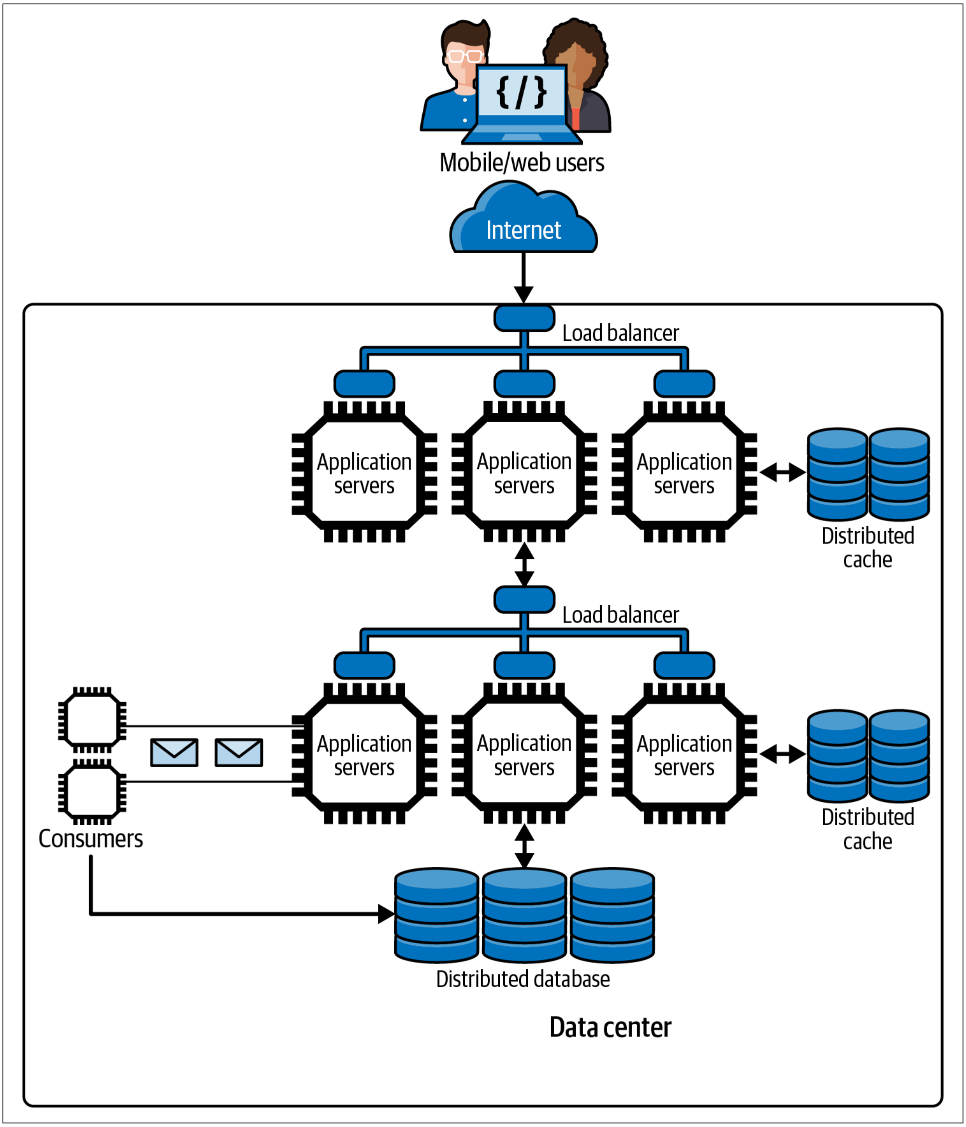

Figure 2-7. Increasing responsiveness with queueing

Whenever the results of a write operation are not immediately needed, an application can use this approach to improve responsiveness and, as a result, scalability. Many queueing technologies exist that applications can utilize, and I'll discuss how these operate in Chapter 7. These queueing platforms all provide asynchronous communications. A *producer* service writes to the queue, which acts as temporary storage, while another *consumer* service removes messages from the queue and makes the necessary updates to, in our example, a database that stores skier lift ride details.

The key is that the data *eventually* gets persisted. Eventually typically means a few seconds at most but use cases that employ this design should be resilient to longer delays without impacting the user experience.

Systems and Hardware Scalability

Even the most carefully crafted software architecture and code will be limited in terms of scalability if the services and data stores run on inadequate hardware. The open source and commercial platforms that are commonly deployed in scalable systems are designed to utilize additional hardware resources in terms of CPU cores, memory, and disks. It's a balancing act between achieving the performance and scalability you require, and keeping your costs as low as possible.

That said, there are some cases where upgrading the number of CPU cores and available memory is not going to buy you more scalability. For example, if code is single threaded, running it on a node with more cores is not going to improve performance. It'll just use one core at any time. The rest are simply not used. If multithreaded code contains many serialized sections, only one threaded core can proceed at a time to ensure the results are correct. This phenomenon is described by Amdahl's law. This gives us a way to calculate the theoretical acceleration of code when adding more CPU cores based on the amount of code that executes serially.

Two data points from Amdahl's law are:

- If only 5% of a code executes serially, the rest in parallel, adding more than 2,048 cores has essentially no effect.
- If 50% of a code executes serially, the rest in parallel, adding more than 8 cores has essentially no effect.

This demonstrates why efficient, multithreaded code is essential to achieving scalability. If your code is not running as highly independent tasks implemented as threads, then not even money will buy you scalability. That's why I devote Chapter 4 to the topic of multithreading—it's a core knowledge component for building scalable distributed systems.

To illustrate the effect of upgrading hardware, Figure 2-8 shows how the throughput of a benchmark system improves as the database is deployed on more powerful (and expensive) hardware.[5] The benchmark employs a Java service that accepts requests from a load generating client, queries a database, and returns the results to the client. The client, service, and database run on different hardware resources deployed in the same regions in the AWS cloud.

5 Results are courtesy of Ruijie Xiao from Northeastern University, Seattle.

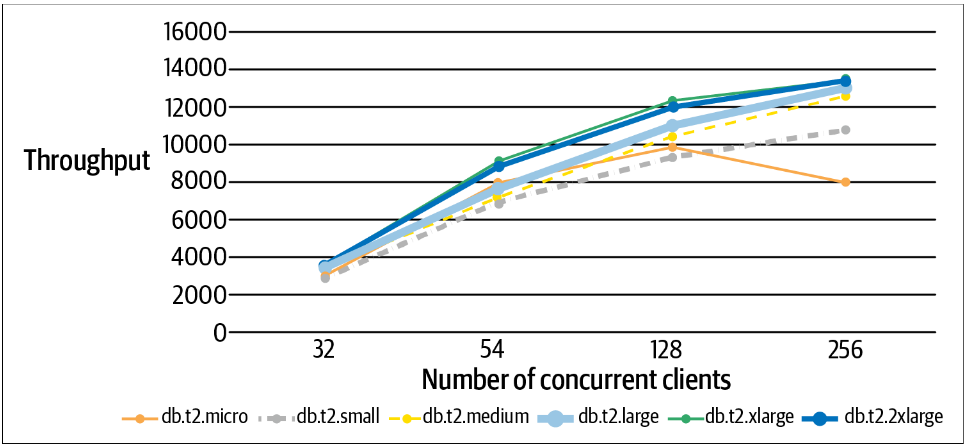

Figure 2-8. An example of scaling up a database server

In the tests, the number of concurrent requests grows from 32 to 256 (*x*-axis) and each line represents the system throughput (*y*-axis) for a different hardware configuration on the AWS EC2's Relational Database Service (RDS). The different configurations are listed at the bottom of the chart, with the least powerful on the left and most powerful on the right. Each client sends a fixed number of requests synchronously over HTTP, with no pause between receiving results from one request and sending the next. This consequently exerts a high request load on the server.

From this chart, it's possible to make some straightforward observations:

- In general, the more powerful the hardware selected for the database, the higher the throughput. That is good.
- The difference between the db.t2.xlarge and db.t2.2xlarge instances in terms of throughput is minimal. This could be because the service tier is becoming a bottleneck, or our database model and queries are not exploiting the additional resources of the db.t2.2xlarge RDS instance. Regardless—more bucks, no bang.
- The two least powerful instances perform pretty well until the request load is increased to 256 concurrent clients. The dip in throughput for these two instances indicates they are overloaded and things will only get worse if the request load increases.

Hopefully, this simple example illustrates why scaling through simple upgrading of hardware needs to be approached carefully. Adding more hardware always increases costs, but may not always give the performance improvement you expect. Running simple experiments and taking measurements is essential for assessing the effects of hardware upgrades. It gives you solid data to guide your design and justify costs to stakeholders.

Summary and Further Reading

In this chapter I've provided a whirlwind tour of the major approaches you can utilize to scale out a system as a collection of communicating services and distributed databases. Much detail has been brushed over, and as you have no doubt realized—in software systems the devil is in the detail. Subsequent chapters will therefore progressively start to explore these details, starting with some fundamental characteristics of distributed systems in Chapter 3 that everyone should be aware of.

Another area this chapter has skirted around is the subject of software architecture. I've used the term *services* for distributed components in an architecture that implement application business logic and database access. These services are independently deployed processes that communicate using remote communications mechanisms such as HTTP. In architectural terms, these services are most closely mirrored by those in the service-oriented architecture (SOA) pattern, an established architectural approach for building distributed systems. A more modern evolution of this approach revolves around microservices. These tend to be more cohesive, encapsulated services that promote continuous development and deployment.

If you'd like a much more in-depth discussion of these issues, and software architecture concepts in general, then Mark Richards' and Neal Ford's book *Fundamentals of Software Architecture: An Engineering Approach* (O'Reilly, 2020) is an excellent place to start.

Finally, there's a class of *big data* software architectures that address some of the issues that come to the fore with very large data collections. One of the most prominent is data reprocessing. This occurs when data that has already been stored and analyzed needs to be reanalyzed due to code or business rule changes. This reprocessing may occur due to software fixes, or the introduction of new algorithms that can derive more insights from the original raw data. There's a good discussion of the Lambda and Kappa architectures, both of which are prominent in this space, in Jay Krepps' 2014 article "Questioning the Lambda Architecture" on the O'Reilly Radar blog.

CHAPTER 3
Distributed Systems Essentials

As I described in Chapter 2, scaling a system naturally involves adding multiple independently moving parts. We run our software components on multiple machines and our databases across multiple storage nodes, all in the quest of adding more processing capacity. Consequently, our solutions are distributed across multiple machines in multiple locations, with each machine processing events concurrently, and exchanging messages over a network.

This fundamental nature of distributed systems has some profound implications on the way we design, build, and operate our solutions. This chapter provides the basic information you need to know to appreciate the issues and complexities of distributed software systems. I'll briefly cover communications networks hardware and software, remote method invocation, how to deal with the implications of communications failures, distributed coordination, and the thorny issue of time in distributed systems.

Communications Basics

Every distributed system has software components that communicate over a network. If a mobile banking app requests the user's current bank account balance, a (very simplified) sequence of communications occurs along the lines of:

1. The mobile banking app sends a request over the cellular network addressed to the bank to retrieve the user's bank balance.
2. The request is routed across the internet to where the bank's web servers are located.

3. The bank's web server authenticates the request (confirms that it originated from the supposed user) and sends a request to a database server for the account balance.
4. The database server reads the account balance from disk and returns it to the web server.
5. The web server sends the balance in a reply message addressed to the app, which is routed over the internet and the cellular network until the balance magically appears on the screen of the mobile device.

It almost sounds simple when you read the above, but in reality, there's a huge amount of complexity hidden beneath this sequence of communications. Let's examine some of these complexities in the following sections.

Communications Hardware

The bank balance request example above will inevitably traverse multiple different networking technologies and devices. The global internet is a heterogeneous machine, comprising different types of network communications channels and devices that shuttle many millions of messages per second across networks to their intended destinations.

Different types of communications channels exist. The most obvious categorization is wired versus wireless. For each category there are multiple network transmission hardware technologies that can ship bits from one machine to another. Each technology has different characteristics, and the ones we typically care about are speed and range.

For physically wired networks, the two most common types are local area networks (LANs) and wide area networks (WANs). LANs are networks that can connect devices at "building scale," being able to transmit data over a small number (e.g., 1–2) of kilometers. Contemporary LANs in data centers can transport between 10 and 100 gigabits per second (Gbps). This is known as the network's bandwidth, or capacity. The time taken to transmit a message across a LAN—the network's latency—is submillisecond with modern LAN technologies.

WANs are networks that traverse the globe and make up what we collectively call the internet. These long-distance connections are the high speed data pipelines connecting cities, countries, and continents with fiber optic cables. These cables support a networking technology known as wavelength division multiplexing which makes it possible to transmit up to 171 Gbps over 400 different channels, giving more than 70 terabits per second (Tbps) of total bandwidth for a single fiber link. The fiber cables that span the world normally comprise four or more strands of fiber, giving bandwidth capacity of hundreds of Tbps for each cable.

Latency is more complicated with WANs, however. WANs transmit data over hundreds and thousands of kilometers, and the maximum speed that the data can travel in fiber optic cables is the theoretical speed of light. In reality, these cables can't reach the speed of light, but do get pretty close to it, as shown in Table 3-1.

Table 3-1. WAN speeds

Path	Distance	Travel time (speed of light)	Travel time (fiber optic cable)
New York to San Francisco	4,148 km	14 ms	21 ms
New York to London	5,585 km	19 ms	28 ms
New York to Sydney	15,993 km	53 ms	80 ms

Actual times will be slower than the fiber optic travel times in Table 3-1 as the data needs to pass through networking equipment known as routers. The global internet has a complex hub-and-spoke topology with many potential paths between nodes in the network. Routers are therefore responsible for transmitting data on the physical network connections to ensure data is transmitted across the internet from source to destination.

Routers are specialized, high-speed devices that can handle several hundred Gbps of network traffic, pulling data off incoming connections and sending the data out to different outgoing network connections based on their destination. Routers at the core of the internet comprise racks of these devices and can process tens to hundreds of Tbps. This is how you and thousands of your friends get to watch a steady video stream on Netflix at the same time.

Wireless technologies have different range and bandwidth characteristics. WiFi routers that we are all familiar with in our homes and offices are wireless Ethernet networks and use 802.11 protocols to send and receive data. The most widely used WiFi protocol, 802.11ac, allows for maximum (theoretical) data rates of up to 5,400 Mbps. The most recent 802.11ax protocol, also known as WiFi 6, is an evolution of 802.11ac technology that promises increased throughput speeds of up to 9.6 Gbps. The range of WiFi routers is of the order of tens of meters and of course is affected by physical impediments like walls and floors.

Cellular wireless technology uses radio waves to send data from our phones to routers mounted on cell towers, which are generally connected by wires to the core internet for message routing. Each cellular technology introduces improved bandwidth and other dimensions of performance. The most common technology at the time of writing is 4G LTE wireless broadband. 4G LTE is around 10 times faster than the older 3G, able to handle sustained download speeds around 10 Mbps (peak download speeds are nearer 50 Mbps) and upload speeds between 2 and 5 Mbps.

Emerging 5G cellular networks promise 10x bandwidth improvements over existing 4G, with 1–2 millisecond latencies between devices and cell towers. This is a great improvement over 4G latencies, which are in the 20–40 millisecond range. The trade-off is range. 5G base station range operates at about 500 meters maximum, whereas 4G provides reliable reception at distances of 10–15 km.

This whole collection of different hardware types for networking comes together in the global internet. The internet is a heterogeneous network, with many different operators around the world and every type of hardware imaginable. Figure 3-1 shows a simplified view of the major components that comprise the internet. Tier 1 networks are the global high-speed internet backbone. There are around 20 Tier 1 internet service providers (ISPs) who manage and control global traffic. Tier 2 ISPs are typically regional (e.g., one country), have lower bandwidth than Tier 1 ISPs, and deliver content to customers through Tier 3 ISPs. Tier 3 ISPs are the ones that charge you exorbitant fees for your home internet every month.

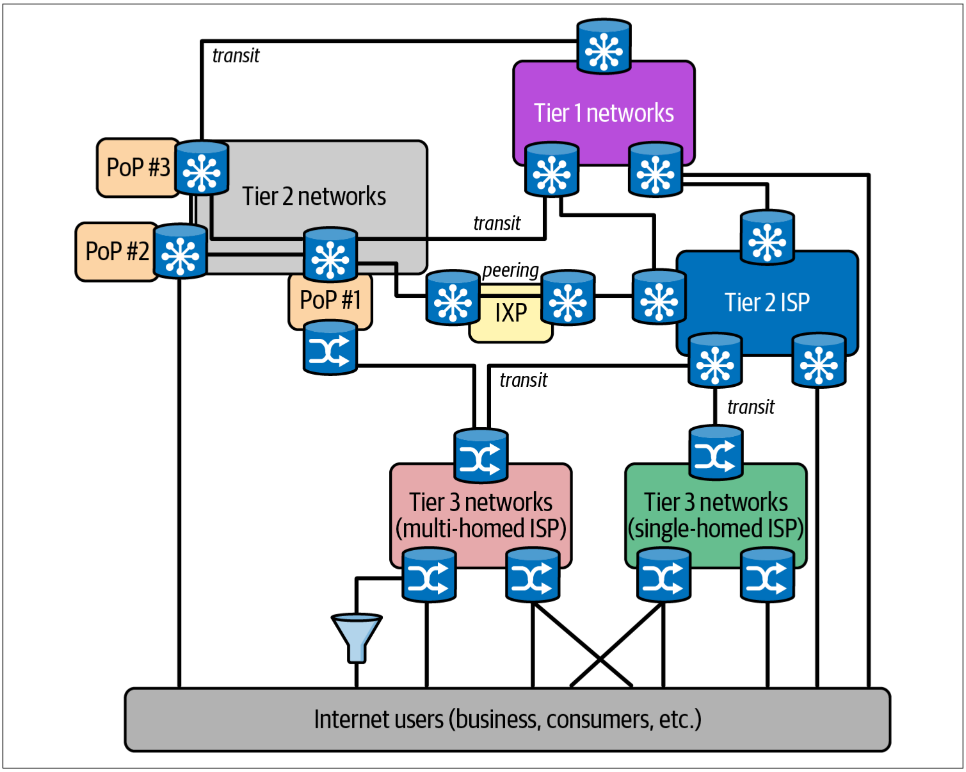

Figure 3-1. Simplified view of the internet

There's a lot more complexity to how the internet works than described here. That level of networking and protocol complexity is beyond the scope of this chapter. From a distributed systems software perspective, we need to understand more about the “magic” that enables all this hardware to route messages from, say, my cell phone to my bank and back. This is where the *Internet Protocol (IP)* comes in.

Communications Software

Software systems on the internet communicate using the Internet Protocol (IP) suite. The IP suite specifies host addressing, data transmission formats, message routing, and delivery characteristics. There are four abstract layers, which contain related protocols that support the functionality required at that layer. These are, from lowest to highest:

1. The data link layer, specifying communication methods for data across a single network segment. This is implemented by the device drivers and network cards that live inside your devices.
2. The internet layer specifies addressing and routing protocols that make it possible for traffic to traverse the independently managed and controlled networks that comprise the internet. This is the IP layer in the internet protocol suite.
3. The transport layer, specifying protocols for reliable and best-effort, host-to-host communications. This is where the well-known Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) live.
4. The application layer, which comprises several application-level protocols such as HTTP and the secure copy protocol (SCP).

Each of the higher-layer protocols builds on the features of the lower layers. In the following section, I’ll briefly cover IP for host discovery and message routing, and TCP and UDP that can be utilized by distributed applications.

Internet Protocol (IP)

IP defines how hosts are assigned addresses on the internet and how messages are transmitted between two hosts who know each other’s addresses.

Every device on the internet has its own address. These are known as Internet Protocol (IP) addresses. The location of an IP address can be found using an internet-wide directory service known as Domain Name System (DNS). DNS is a widely distributed, hierarchical database that acts as the address book of the internet.

The technology currently used to assign IP addresses, known as Internet Protocol version 4 (IPv4), will eventually be replaced by its successor, IPv6. IPv4 is a 32-bit addressing scheme that before long will run out of addresses due to the number of devices connecting to the internet. IPv6 is a 128-bit scheme that will offer an (almost) infinite number of IP addresses. As an indicator, in July 2020 about 33% of the traffic processed by Google.com is IPv6.

DNS servers are organized hierarchically. A small number of root DNS servers, which are highly replicated, are the starting point for resolving an IP address. When an internet browser tries to find a website, a network host known as the local DNS server (managed by your employer or ISP) will contact a root DNS server with the requested hostname. The root server replies with a referral to a so-called *authoritative* DNS server that manages name resolution for, in our banking example, *.com* addresses. There is an authoritative name server for each top-level internet domain (*.com*, *.org*, *.net*, etc.).

Next, the local DNS server will query the *.com* DNS server, which will reply with the address of the DNS server that knows about all the IP addresses managed by *igbank.com*. This DNS is queried, and it returns the actual IP address we need to communicate with the application. The overall scheme is illustrated in Figure 3-2.

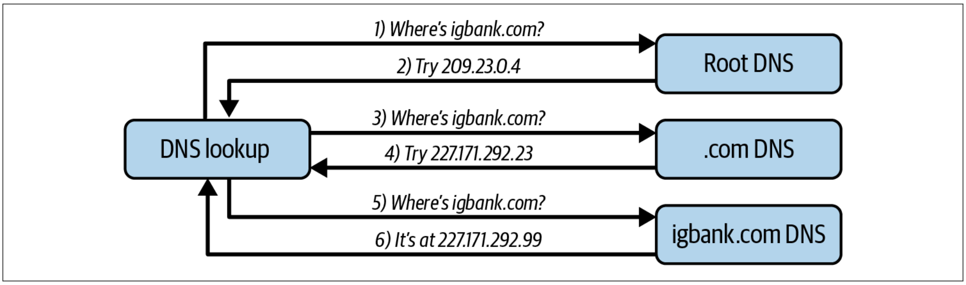

Figure 3-2. Example DNS lookup for igbank.com

The whole DNS database is highly geographically replicated so there are no single points of failure, and requests are spread across multiple physical servers. Local DNS servers also remember the IP addresses of recently contacted hosts, which is possible as IP addresses don't change very often. This means the complete name resolution process doesn't occur for every site we contact.

Armed with a destination IP address, a host can start sending data across the network as a series of IP data packets. IP delivers data from the source to the destination host based on the IP addresses in the packet headers. IP defines a packet structure that contains the data to be delivered, along with header data including source and destination IP addresses. Data sent by an application is broken up into a series of packets which are independently transmitted across the internet.

IP is known as a best-effort delivery protocol. This means it does not attempt to compensate for the various error conditions that can occur during packet transmission. Possible transmission errors include data corruption, packet loss, and duplication. In addition, every packet is routed across the internet from source to destination independently. Treating every packet independently is known as packet switching. This allows the network to dynamically respond to conditions such as network link failure and congestion, and hence is a defining characteristic of the internet. This does mean, however, that different packets may be delivered to the same destination via different network paths, resulting in out-of-order delivery to the receiver.

Because of this design, the IP is unreliable. If two hosts require reliable data transmission, they need to add additional features to make this occur. This is where the next layer in the IP protocol suite, the transport layer, enters the scene.

Transmission Control Protocol (TCP)

Once an application or browser has discovered the IP address of the server it wishes to communicate with, it can send messages using a transport protocol API. This is achieved using TCP or UDP, which are the popular standard transport protocols for the IP network stack.

Distributed applications can choose which of these protocols to use. Implementations are widely available in mainstream programming languages such as Java, Python, and C++. In reality, use of these APIs is not common as higher-level programming abstractions hide the details from most applications. In fact, the IP protocol suite application layer contains several of these application-level APIs, including HTTP, which is very widely used in mainstream distributed systems.

Still, it's important to understand TCP, UDP, and their differences. Most requests on the internet are sent using TCP. TCP is:

- Connection-oriented
- Stream-oriented
- Reliable

I'll explain each of these qualities, and why they matter, below.

TCP is known as a connection-oriented protocol. Before any messages are exchanged between applications, TCP uses a three-step handshake to establish a two-way connection between the client and server applications. The connection stays open until the TCP client calls `close()` to terminate the connection with the TCP server. The server responds by acknowledging the `close()` request before the connection is dropped.

Once a connection is established, a client sends a sequence of requests to the server as a data stream. When a data stream is sent over TCP, it is broken up into individual network packets, with a maximum packet size of 65,535 bytes. Each packet contains a source and destination address, which is used by the underlying IP protocol to route the messages across the network.

The internet is a packet switched network, which means every packet is individually routed across the network. The route each packet traverses can vary dynamically based on the conditions in the network, such as link congestion or failure. This means the packets may not arrive at the server in the same order they are sent from the client. To solve this problem, a TCP sender includes a sequence number in each packet so the receiver can reassemble packets into a stream that is identical to the order they were sent.

Reliability is needed as network packets can be lost or delayed during transmission between sender and receiver. To achieve reliable packet delivery, TCP uses a cumulative acknowledgment mechanism. This means a receiver will periodically send an acknowledgment packet that contains the highest sequence number of the packets received without gaps in the packet stream. This implicitly acknowledges all packets sent with a lower sequence number, meaning all have been successfully received. If a sender doesn't receive an acknowledgment within a timeout period, the packet is resent.

TCP has many other features, such as checksums to check packet integrity, and dynamic flow control to ensure a sender doesn't overwhelm a slow receiver by sending data too quickly. Along with connection establishment and acknowledgments, this makes TCP a relatively heavyweight protocol, which trades off reliability over efficiency.

This is where UDP comes into the picture. UDP is a simple, connectionless protocol, which exposes the user's program to any unreliability of the underlying network. There is no guarantee that delivery will occur in a prescribed order, or that it will happen at all. It can be thought of as a thin veneer (layer) on top of the underlying IP protocol, and deliberately trades off raw performance over reliability.

This, however, is highly appropriate for many modern applications where the odd lost packet has very little effect. Think streaming movies, video conferencing, and gaming, where one lost packet is unlikely to be perceptible by a user.

Figure 3-3 depicts some of the major differences between TCP and UDP. TCP incorporates a connection establishment three-packet handshake (SYN, SYN ACK, ACK), and piggybacks acknowledgments (ACK) of packets so that any packet loss can be handled by the protocol. There's also a TCP connection close phase involving a four-way handshake that is not shown in the diagram. UDP dispenses with connection establishment, tear down, acknowledgments, and retries. Therefore, applications using UDP need to be tolerant of packet loss and client or server failures (and behave accordingly).

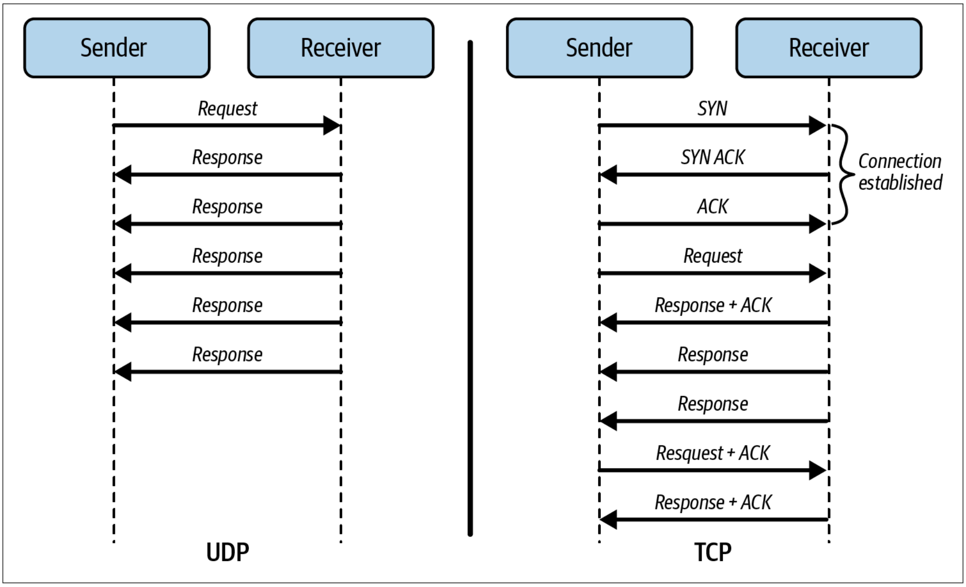

Figure 3-3. Comparing TCP and UDP

Remote Method Invocation

It's perfectly feasible to write our distributed applications using low-level APIs that interact directly with the transport layer protocols TCP and UDP. The most common approach is the standardized sockets library—see the brief overview in the sidebar. This is something you'll hopefully never need to do, as sockets are complex and error prone. Essentially, sockets create a bidirectional pipe between two nodes that you can use to send streams of data. There are (luckily) much better ways to build distributed communications, as I'll describe in this section. These approaches abstract away much of the complexity of using sockets. However, sockets still lurk underneath, so some knowledge is necessary.

An Overview of Sockets

A socket is one endpoint of a two-way network connection between a client and a server. Sockets are identified by a combination of the node's IP address and an abstraction known as a *port*. A port is a unique numeric identifier, which allows a node to support communications for multiple applications running on the node.

Each IP address can support 65,535 TCP ports and another 65,535 UDP ports. On a server, each *{<IP Address>, <port>}* combination can be associated with an application. This combination forms a unique endpoint that the transport layer uses to deliver data to the desired server.

A socket connection is identified by a unique combination of client and server IP addresses and ports, namely <*client IP address, client port, server IP address, server port*>. Each unique connection also allocates a socket descriptor on both the client and the server. Once the connection is created, the client sends data to the server in a stream, and the server responds with results. The sockets library supports both protocols, with the `SOCK_STREAM` option for TCP, and the `SOCK_DGRAM` for UDP.

You can write distributed applications directly to the sockets API, which is an operating system core component. Socket APIs are available in all mainstream programming languages. However, the sockets library is a low-level, hard-to-use API. You should avoid it unless you have a real need to write system-level code.

In our mobile banking example, the client might request a balance for the user's checking account using sockets. Ignoring specific language issues (and security!), the client could send a message payload as follows over a connection to the server:

{"balance", "000169990"}

In this message, "balance" represents the operation we want the server to execute, and "000169990" is the bank account number.

In the server, we need to know that the first string in the message is the operation identifier, and based on this value being "balance", the second is the bank account number. The server then uses these values to presumably query a database, retrieve the balance, and send back the results, perhaps as a message formatted with the account number and current balance, as below:

{"000169990", "220.77"}

In any complex system, the server will support many operations. In *igbank.com*, there might be for example "login", "transfer", "address", "statement", "transactions", and so on. Each will be followed by different message payloads that the server needs to interpret correctly to fulfill the client's request.

What we are defining here is an application-specific protocol. As long as we send the necessary values in the correct order for each operation, the server will be able to respond correctly. If we have an erroneous client that doesn't adhere to our application protocol, well, our server needs to do thorough error checking. The socket library provides a primitive, low-level method for client/server communications. It provides highly efficient communications but is difficult to correctly implement and evolve the application protocol to handle all possibilities. There are better mechanisms.

Stepping back, if we were defining the *igbank.com* server interface in an object-oriented language such as Java, we would have each operation it can process as a method. Each method is passed an appropriate parameter list for that operation, as shown in this example code:

```
// Simple igbank.com server interface
public interface IGBank {
    public float balance  (String accNo);
    public boolean  statement(String month) ;
    // other operations
}
```

There are several advantages of having such an interface, namely:

- Calls from the client to the server can be statically checked by the compiler to ensure they are of the correct format and argument types.
- Changes in the server interface (e.g., adding a new parameter) force changes in the client code to adhere to the new method signature.
- The interface is clearly defined by the class definition and thus straightforward for a client programmer to understand and utilize.

These benefits of an explicit interface are of course well known in software engineering. The whole discipline of object-oriented design is pretty much based upon these foundations, where an interface defines a contract between the caller and callee. Compared to the implicit application protocol we need to follow with sockets, the advantages are significant.

This fact was recognized reasonably early in the creation of distributed systems. Since the early 1990s, we have seen an evolution of technologies that enable us to define explicit server interfaces and call these across the network using essentially the same syntax as we would in a sequential program. A summary of the major approaches is given in Table 3-2. Collectively, they are known as Remote Procedure Call (RPC), or Remote Method Invocation (RMI) technologies.

Table 3-2. Summary of major RPC/RMI technologies

Technology	Date	Main features
Distributed Computing Environment (DCE)	Early 1990s	DCE RPC provides a standardized approach for client/server systems. Primary languages were C/C++.
Common Object Request Broker Architecture (CORBA)	Early 1990s	Facilitates language-neutral client/server communications based on an object-oriented interface definition language (IDL). Primary language support in C/C++, Java, Python, and Ada.
Java Remote Method Invocation (RMI)	Late 1990s	A pure Java-based remote method invocation that facilitates distributed client/server systems with the same semantics as Java objects.
XML web services	2000	Supports client/server communications based on HTTP and XML. Servers define their remote interface in the Web Services Description Language (WSDL).
gRPC	2015	Open source, based on HTTP/2 for transport, and uses Protocol Buffers (Protobuf) as the interface description language

While the syntax and semantics of these RPC/RMI technologies vary, the essence of how each operates is the same. Let's continue with our Java example of *igbank.com* to examine the whole class of approaches. Java offers a Remote Method Invocation (RMI) API for building client/server applications.

Using Java RMI, we can trivially make our `IGBank` interface example from above into a remote interface, as illustrated in the following code:

```
import java.rmi.*;
// Simple igbank.com server interface
public interface IGBank extends Remote{
    public float balance  (String accNo)
         throws RemoteException;
    public boolean  statement(String month)
         throws RemoteException ;
    // other operations
 }
```

The `java.rmi.Remote` interface serves as a marker to inform the Java compiler we are creating an RMI server. In addition, each method must throw `java.rmi.RemoteException`. These exceptions represent errors that can occur when a distributed call between two objects is invoked over a network. The most common reasons for such an exception would be a communications failure or the server object having crashed.

We then must provide a class that implements this remote interface. The sample code below shows an extract of the server implementation:

```
public class IGBankServer extends UnicastRemoteObject
                          implements IGBank  {
   // constructor/method implementations omitted
   public static void main(String args[]){
        try{
          IGBankServer server=new IGBankServer();
          // create a registry in local JVM on default port
          Registry registry = LocateRegistry.createRegistry(1099);
          registry.bind("IGBankServer", server);
          System.out.println("server ready");
        }catch(Exception e){
                 // code omitted for brevity}
        }
   }
```

Points to note are:

- The server extends the `UnicastRemoteObject` class. This essentially provides the functionality to instantiate a remotely callable object.
- Once the server object is constructed, its availability must be advertised to remote clients. This is achieved by storing a reference to the object in a system service known as the *RMI registry*, and associating a logical name with it—in this example, "*IGBankServer*." The registry is a simple directory service that enables clients to look up the location (network address and object reference) of and

obtain a reference to an RMI server by simply supplying the logical name it is associated with in the registry.

An extract from the client code to connect to the server is shown in the following example. It obtains a reference to the remote object by performing a `lookup` operation in the RMI registry and specifying the logical name that identifies the server. The reference returned by the lookup operation can then be used to call the server object in the same manner as a local object. However, there is a difference—the client must be ready to catch a `RemoteException` that will be thrown by the Java runtime when the server object cannot be reached:

```
// obtain a remote reference to the server
IGBank bankServer=
        (IGBank)Naming.lookup("rmi://localhost:1099/IGBankServer");
//now we can call the server
System.out.println(bankServer.balance("00169990"));
```

Figure 3-4 depicts the call sequence among the components that comprise an RMI system. The `Stub` and `Skeleton` are objects generated by the compiler from the RMI interface definition, and these facilitate the actual remote communications. The skeleton is in fact a TCP network endpoint (*host, port*) that listens for calls to the associated server.

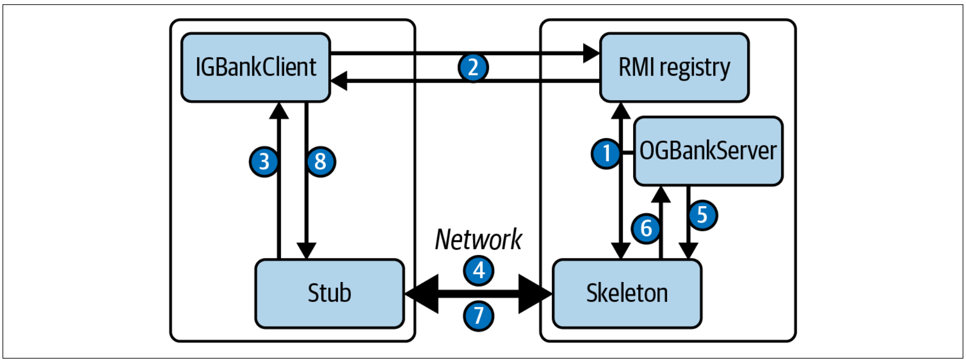

Figure 3-4. Schematic depicting the call sequence for establishing a connection and making a call to an RMI server object

The sequence of operations is as follows:

1. When the server starts, its logical reference is stored in the RMI registry. This entry contains the Java client stub that can be used to make remote calls to the server.
2. The client queries the registry, and the stub for the server is returned.
3. The client stub accepts a method call to the server interface from the Java client implementation.

4. The stub transforms the request into one or more network packets that are sent to the server host. This transformation process is known as marshalling.
5. The skeleton accepts network requests from the client, and unmarshalls the network packet data into a valid call to the RMI server object implementation. Unmarshalling is the opposite of marshalling—it takes a sequence of network packets and transforms them into a call to an object.
6. The skeleton waits for the method to return a response.
7. The skeleton marshalls the method results into a network reply packet that is returned to the client.
8. The stub unmarshalls the data and passes the result to the Java client call site.

This Java RMI example illustrates the basics that are used for implementing any RPC/RMI mechanism, even in modern languages like Erlang and Go. You are most likely to encounter Java RMI when using the Java Enterprise JavaBeans (EJB) technology. EJBs are a server-side component model built on RMI, which have seen wide usage in the last 20 or so years in enterprise systems.

Regardless of the precise implementation, the basic attraction of RPC/RMI approaches is to provide an abstract calling mechanism that supports *location transparency* for clients making remote server calls. Location transparency is provided by the registry, or in general any mechanism that enables a client to locate a server through a directory service. This means it is possible for the server to update its network location in the directory without affecting the client implementation.

RPC/RMI is not without its flaws. Marshalling and unmarshalling can become inefficient for complex object parameters. Cross-language marshalling—client in one language, server in another—can cause problems due to types being represented differently in different languages, causing subtle incompatibilities. And if a remote method signature changes, all clients need to obtain a new compatible stub, which can be cumbersome in large deployments.

For these reasons, most modern systems are built around simpler protocols based on HTTP and using JSON for parameter representation. Instead of operation names, HTTP verbs (`PUT`, `GET`, `POST`, etc.) have associated semantics that are mapped to a specific URL. This approach originated in the work by Roy Fielding on the REST approach.[1] REST has a set of semantics that comprise a *RESTful* architecture style, and in reality most systems do not adhere to these. We'll discuss REST and HTTP API mechanisms in Chapter 5.

1 Roy T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures." Dissertation, University of California, Irvine, 2000.

Partial Failures

The components of distributed systems communicate over a network. In communications technology terminology, the shared local and wide area networks that our systems communicate over are known as *asynchronous* networks.

With asynchronous networks:

- Nodes can choose to send data to other nodes at any time.
- The network is *half-duplex*, meaning that one node sends a request and must wait for a response from the other. These are two separate communications.
- The time for data to be communicated between nodes is variable, due to reasons like network congestion, dynamic packet routing, and transient network connection failures.
- The receiving node may not be available due to a software or machine crash.
- Data can be lost. In wireless networks, packets can be corrupted and hence dropped due to weak signals or interference. Internet routers can drop packets during congestion.
- Nodes do not have identical internal clocks; hence they are not synchronized.

> This is in contrast with synchronous networks, which essentially are full duplex, transmitting data in both directions at the same time with each node having an identical clock for synchronization.

What does this mean for our applications? Well, put simply, when a client sends a request to a server, how long does it wait until it receives a reply? Is the server node just being slow? Is the network congested and the packet has been dropped by a router? If the client doesn't get a reply, what should it do?

Let's explore these scenarios in detail. The core problem here, namely whether and when a response is received, is known as handling partial failures, and the general situation is depicted in Figure 3-5.

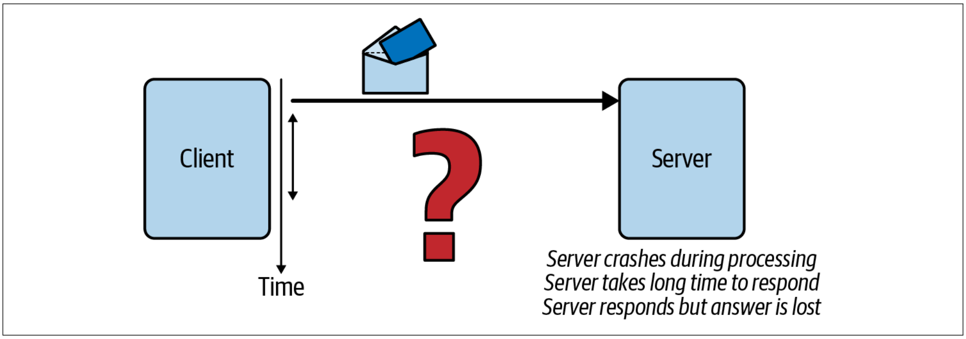

Figure 3-5. Handling partial failures

When a client wishes to connect to a server and exchanges messages, the following outcomes may occur:

- The request succeeds and a rapid response is received. All is well. (In reality, this outcome occurs for almost every request. *Almost* is the operative word here.)
- The destination IP address lookup may fail. In this case, the client rapidly receives an error message and can act accordingly.
- The IP address is valid but the destination node or target server process has failed. The sender will receive a timeout error message and can inform the user.
- The request is received by the target server, which fails while processing the request and no response is ever sent.
- The request is received by the target server, which is heavily loaded. It processes the request but takes a long time (e.g., 34 seconds) to respond.
- The request is received by the target server and a response is sent. However, the response is not received by the client due to a network failure.

The first three points are easy for the client to handle, as a response is received rapidly. A result from the server or an error message—either allows the client to proceed. Failures that can be detected quickly are easy to deal with.

The rest of the outcomes pose a problem for the client. They do not provide any insight into the reason why a response has not been received. From the client's perspective, these three outcomes look exactly the same. The client cannot know without waiting (potentially forever) whether the response will arrive eventually or never arrive; waiting forever doesn't get much work done.

More insidiously, the client cannot know if the operation succeeded and a server or network failure caused the result to never arrive, or if the request is on its way—delayed simply due to congestion in the network/server. These faults are collectively known as *crash faults*.

The typical solution that clients adopt to handle crash faults is to resend the request after a configured timeout period. However, this is fraught with danger, as Figure 3-6 illustrates. The client sends a request to the server to deposit money in a bank account. When it receives no response after a timeout period, it resends the request. What is the resulting balance? The server may have applied the deposit, or it may not, depending on the partial failure scenario.

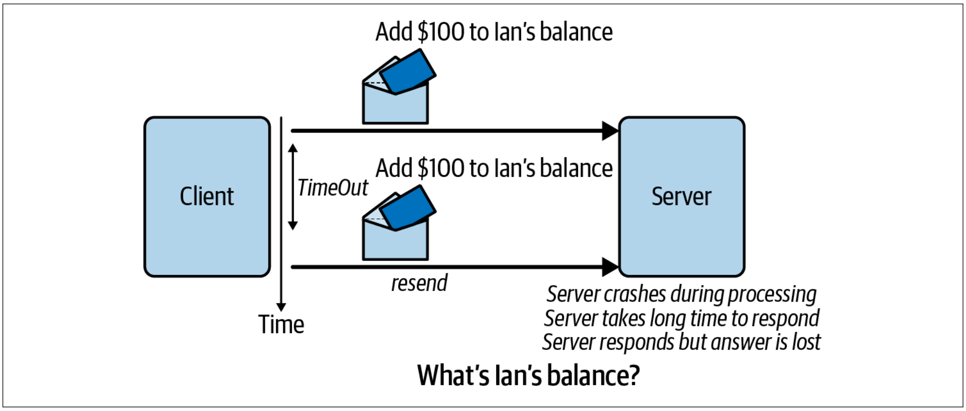

Figure 3-6. Client retries a request after timeout

The chance that the deposit may occur twice is a fine outcome for the customer, but the bank is unlikely to be amused by this possibility. Therefore, we need a way to ensure in our server operations implementation that retried, duplicate requests from clients only result in the request being applied once. This is necessary to maintain correct application semantics.

This property is known as *idempotence*. Idempotent operations can be applied multiple times without changing the result beyond the initial application. This means that for the example in Figure 3-6, the client can retry the request as many times as it likes, and the account will only be increased by $100.

Requests that make no persistent state changes are naturally idempotent. This means all read requests are inherently safe and no extra work is needed on the server. Updates are a different matter. The system needs to devise a mechanism such that duplicate client requests do not cause any state changes and can be detected by the server. In API terms, these endpoints cause mutation of the server state and must therefore be idempotent.

The general approach to building idempotent operations is as follows:

- Clients include a unique *idempotency key* in all requests that mutate state. The key identifies a single operation from the specific client or event source. It is usually a composite of a user identifier, such as the session key, and a unique value such as a local timestamp, UUID, or a sequence number.

- When the server receives a request, it checks to see if it has previously seen the idempotency key value by reading from a database that is uniquely designed for implementing idempotence. If the key is not in the database, this is a new request. The server therefore performs the business logic to update the application state. It also stores the idempotency key in a database to indicate that the operation has been successfully applied.
- If the idempotency key is in the database, this indicates that this request is a retry from the client and hence should not be processed. In this case the server returns a valid response for the operation so that (hopefully) the client won't retry again.

The database used to store idempotency keys can be implemented in, for example:

- A separate database table or collection in the transactional database used for the application data
- A dedicated database that provides very low latency lookups, such as a simple key-value store

Unlike application data, idempotency keys don't have to be retained forever. Once a client receives an acknowledgment of a success for an individual operation, the idempotency key can be discarded. The simplest way to achieve this is to automatically remove idempotency keys from the store after a specific time period, such as 60 minutes or 24 hours, depending on application needs and request volumes.

In addition, an idempotent API implementation must ensure that the application state is modified *and* the idempotency key is stored. Both must occur for success. If the application state is modified and, due to some failure, the idempotent key is not stored, then a retry will cause the operation to be applied twice. If the idempotency key is stored but for some reason the application state is not modified, then the operation has not been applied. If a retry arrives, it will be filtered out as duplicate as the idempotency key already exists, and the update will be lost.

The implication here is that the updates to the application state and idempotency key store must *both* occur, or *neither* must occur. If you know your databases, you'll recognize this as a requirement for transactional semantics. We'll discuss how distributed transactions are achieved in Chapter 12. Essentially, transactions ensure *exactly-once semantics for operations*, which guarantees that all messages will always be processed exactly once—precisely what we need for idempotence.

Exactly once does not mean that there are no message transmission failures, retries, and application crashes. These are all inevitable. The important thing is that the retries eventually succeed and the result is always the same.

We'll return to the issue of communications delivery guarantees in later chapters. As Figure 3-7 illustrates, there's a spectrum of semantics, each with different guarantees and performance characteristics. *At-most-once* delivery is fast and unreliable—this is what the UDP protocol provides. *At-least-once* delivery is the guarantee provided by TCP/IP, meaning duplicates are inevitable. *Exactly-once* delivery, as we've discussed here, requires guarding against duplicates and hence trades off reliability against slower performance.

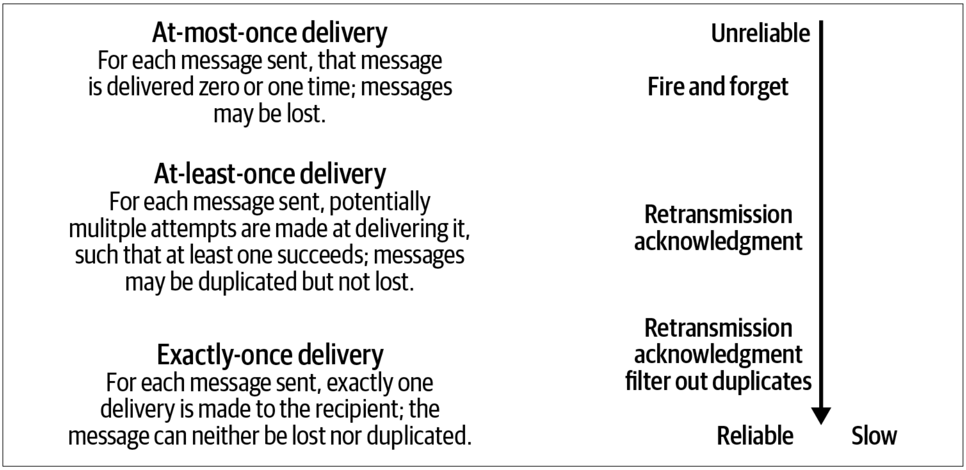

Figure 3-7. Communications delivery guarantees

As we'll see, some advanced communications mechanisms can provide our applications with exactly-once semantics. However, these don't operate at internet scale because of the performance implications. That is why, as our applications are built on the at-least-once semantics of TCP/IP, we must implement exactly-once semantics in our APIs that cause state mutation.

Consensus in Distributed Systems

Crash faults have another implication for the way we build distributed systems. This is best illustrated by the Two Generals' Problem, which is depicted in Figure 3-8.

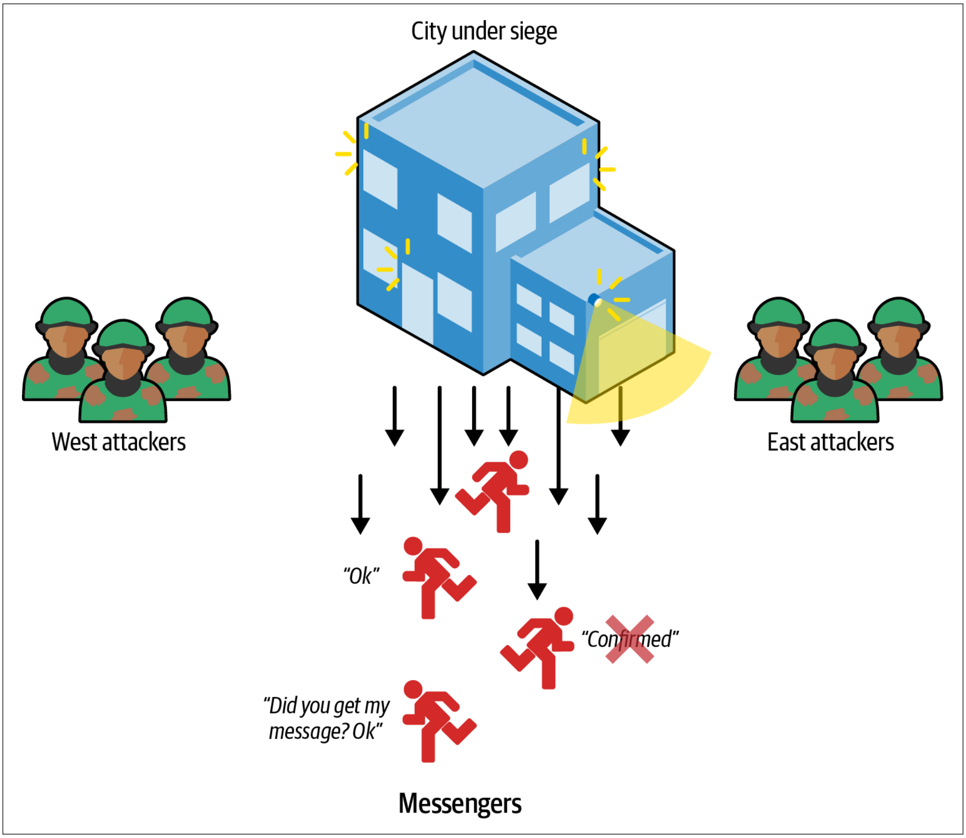

Figure 3-8. The Two Generals' Problem

Imagine a city under siege by two armies. The armies lie on opposite sides of the city, and the terrain surrounding the city is difficult to travel through and visible to snipers in the city. In order to overwhelm the city, it's crucial that both armies attack at the same time. This will stretch the city's defenses and make victory more likely for the attackers. If only one army attacks, then they will likely be repelled.

Given these constraints, how can the two generals reach agreement on the exact time to attack, such that both generals know for certain that agreement has been reached? They both need certainty that the other army will attack at the agreed time, or disaster will ensue.

To coordinate an attack, the first general sends a messenger to the other, with instructions to attack at a specific time. As the messenger may be captured or killed by snipers, the sending general cannot be certain the message has arrived unless they get an acknowledgment messenger from the second general. Of course, the acknowledgment messenger may be captured or killed, so even if the original messenger does get through, the first general may never know. And even if the acknowledgment message arrives, how does the second general know this, unless they get an acknowledgment from the first general?

Hopefully the problem is apparent. With messengers being randomly captured or extinguished, there is no guarantee the two generals will ever reach consensus on the attack time. In fact, it can be proven that it is not possible to *guarantee* agreement will be reached. There are solutions that increase the likelihood of reaching consensus. For example, *Game of Thrones* style, each general may send 100 different messengers every time, and even if most are killed, this increases the probability that at least one will make the perilous journey to the other friendly army and successfully deliver the message.

The Two Generals' Problem is analogous to two nodes in a distributed system wishing to reach agreement on some state, such as the value of a data item that can be updated at either. Partial failures are analogous to losing messages and acknowledgments. Messages may be lost or delayed for an indeterminate period of time—the characteristics of asynchronous networks, as I described earlier in this chapter.

In fact it can be demonstrated that consensus on an asynchronous network in the presence of crash faults, where messages can be delayed but not lost, is impossible to achieve within bounded time. This is known as the FLP Impossibility Theorem.[2]

Luckily, this is only a theoretical limitation, demonstrating it's not possible to *guarantee* consensus will be reached with unbounded message delays on an asynchronous network. In reality, distributed systems reach consensus all the time. This is possible because while our networks are asynchronous, we can establish sensible practical bounds on message delays and retry after a timeout period. FLP is therefore a worst-case scenario, and as such I'll discuss algorithms for establishing consensus in distributed databases in Chapter 12.

Finally, we should note the issue of Byzantine failures. Imagine extending the Two Generals' Problem to *N* generals who need to agree on a time to attack. However, in this scenario, traitorous messengers may change the value of the time of the attack, or a traitorous general may send false information to other generals.

This class of *malicious* failures are known as Byzantine faults and are particularly sinister in distributed systems. Luckily, the systems we discuss in this book typically live behind well-protected, secure enterprise networks and administrative environments. This means we can in practice exclude handling Byzantine faults. Algorithms that do address such malicious behaviors exist, and if you are interested in a practical example, take a look at blockchain consensus mechanisms and Bitcoin.

2 Michael J. Fischer et al., "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM* 32, no. 2 (1985): 374–82. *https://doi.org/10.1145/3149.214121*.

Time in Distributed Systems

Every node in a distributed system has its own internal clock. If all the clocks on every machine were perfectly synchronized, we could always simply compare the timestamps on events across nodes to determine the precise order they occurred in. If this were reality, many of the problems I'll discuss with distributed systems would pretty much go away.

Unfortunately, this is not the case. Clocks on individual nodes *drift* due to environmental conditions like changes in temperature or voltage. The amount of drift varies on every machine, but values such as 10–20 seconds per day are not uncommon. (Or with my current coffee machine at home, about 5 minutes per day!)

If left unchecked, clock drift would render the time on a node meaningless—like the time on my coffee machine if I don't correct it every few days. To address this problem, a number of *time services* exist. A time service represents an accurate time source, such as a GPS or atomic clock, which can be used to periodically reset the clock on a node to correct for drift on packet-switched, variable-latency data networks.

The most widely used time service is Network Time Protocol (NTP), which provides a hierarchically organized collection of time servers spanning the globe. The root servers, of which there are around 300 worldwide, are the most accurate. Time servers in the next level of the hierarchy (approximately 20,000) synchronize to within a few milliseconds of the root server periodically, and so on throughout the hierarchy, with a maximum of 15 levels. Globally, there are more than 175,000 NTP servers.

Using the NTP protocol, a node in an application running an NTP client can synchronize to an NTP server. The time on a node is set by a UDP message exchange with one or more NTP servers. Messages are time stamped, and through the message exchange the time taken for message transit is estimated. This becomes a factor in the algorithm used by NTP to establish what the time on the client should be reset to. A simple NTP configuration is shown in Figure 3-9. On a LAN, machines can synchronize to an NTP server within a small number of milliseconds accuracy.

One interesting effect of NTP synchronization for our applications is that the resetting of the clock can move the local node time forward or backward. This means that if our application is measuring the time taken for events to occur (e.g., to calculate event response times), it is possible that the end time of the event may be earlier than the start time if the `NTP` protocol has set the local time backward.

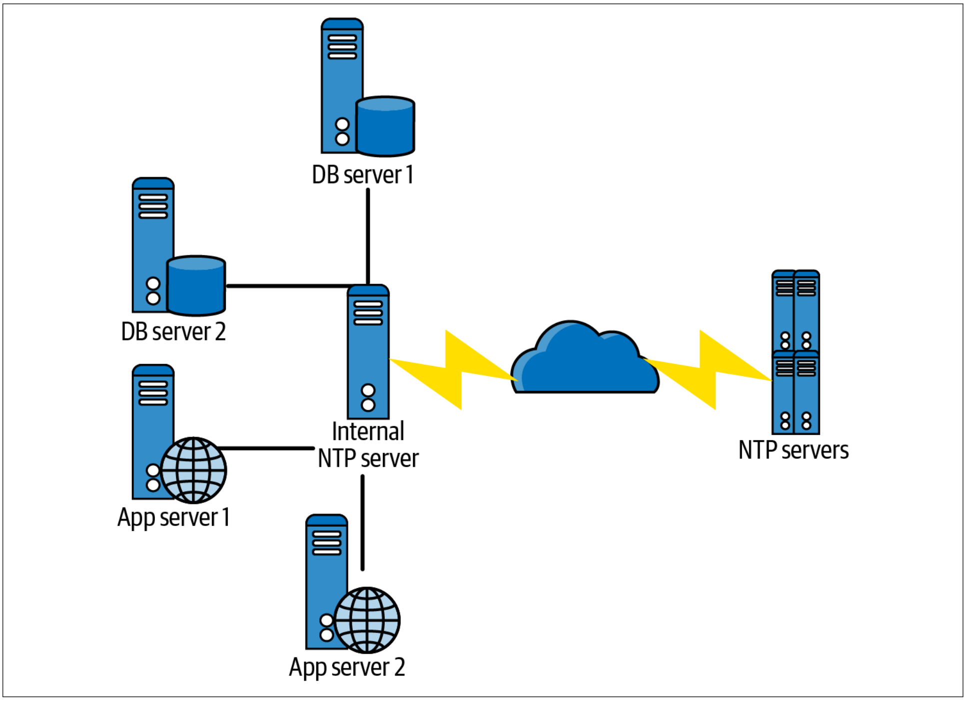

Figure 3-9. Illustrating using the NTP service

In fact, a compute node has two clocks. These are:

Time of day clock
: This represents the number of milliseconds since midnight, January 1st 1970. In Java, you can get the current time using `System.currentTimeMillis()`. This is the clock that can be reset by NTP, and hence may jump forward or backward if it is a long way behind or ahead of NTP time.

Monotonic clock
: This represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past, such as the last time the system was restarted. It will only ever move forward; however, it again may not be a totally accurate measure of elapsed time because it stalls during an event such as virtual machine suspension. In Java, you can get the current monotonic clock time using `System.nanoTime()`.

Applications can use an NTP service to ensure the clocks on every node in the system are closely synchronized. It's typical for an application to resynchronize clocks on anything from a one hour to one day time interval. This ensures the clocks remain close in value. Still, if an application really needs to precisely know the order of events that occur on different nodes, clock drift is going to make this fraught with danger.

There are other time services that provide higher accuracy than NTP. Chrony supports the NTP protocol but provides much higher accuracy and greater scalability than NTP—the reason it has been adopted by Facebook. Amazon has built the Amazon Time Sync Service by installing GPS and atomic clocks in its data centers. This service is available for free to all AWS customers.

The takeaway from this discussion is that our applications cannot rely on timestamps of events on different nodes to represent the actual order of these events. Clock drift even by a second or two makes cross-node timestamps meaningless to compare. The implications of this will become clear when we start to discuss distributed databases in detail.

Summary and Further Reading

This chapter has covered a lot of ground to explain some of the essential characteristics of communications and time in distributed systems. These characteristics are important for application designers and developers to understand.

The key issues that should resonate from this chapter are as follows:

1. Communications in distributed systems can transparently traverse many different types of underlying physical networks, including WiFi, wireless, WANs, and LANs. Communication latencies are hence highly variable, and influenced by the physical distance between nodes, physical network properties, and transient network congestion. At large scale, latencies between application components are something that should be minimized as much as possible (within the laws of physics, of course).
2. The Internet Protocol stack ensures reliable communications across heterogeneous networks through a combination of the IP and TCP protocols. Communications can fail due to network communications fabric and router failures that make nodes unavailable, as well as individual node failure. Your code will experience various TCP/IP overheads, for example, for connection establishment, and errors when network failures occur. Hence, understanding the basics of the IP suite is important for design and debugging.
3. RMI/RPC technologies build the TCP/IP layer to provide abstractions for client/server communications that mirror making local method/procedure calls. However, these more abstract programming approaches still need to be resilient to network issues such as failures and retransmissions. This is most apparent in application APIs that mutate state on the server, and must be designed to be idempotent.
4. Achieving agreement, or consensus on state across multiple nodes in the presence of crash faults is not possible in bounded time on asynchronous networks. Luckily, real networks, especially LANs, are fast and mostly reliable, meaning we

can devise algorithms that achieve consensus in practice. I'll cover these in Part III of the book when we discuss distributed databases.

5. There is no reliable global time source that nodes in an application can rely upon to synchronize their behavior. Clocks on individual nodes vary and cannot be used for meaningful comparisons. This means applications cannot meaningfully compare clocks on different nodes to determine the order of events.

These issues will pervade the discussions in the rest of this book. Many of the unique problems and solutions that are adopted in distributed systems stem from these fundamentals. There's no escaping them!

An excellent source for more detailed, more theoretical coverage of all aspects of distributed systems is George Coulouris et al., *Distributed Systems: Concepts and Design*, 5th ed. (Pearson, 2001).

Likewise for computer networking, you'll find out all you wanted to know and no doubt more in James Kurose and Keith Ross's *Computer Networking: A Top-Down Approach*, 7th ed. (Pearson, 2017).

About the Author

Ian Gorton has 30 years' experience as a software architect, author, computer science professor, and consultant. He has focused on distributed technologies since his days in graduate school and has worked on large-scale software systems in areas such as banking, telecommunications, government, health care, and scientific modeling and simulation. During this time, he has seen software systems evolve to the massive scale they routinely operate at today.

Ian has written three books, including *Essential Software Architecture* and *Data Intensive Computing*, and is the author of over 200 scientific and professional publications on software architecture and software engineering. At the Carnegie Mellon Software Engineering Institute, he led R&D projects in big data and massively scalable systems, and he has continued working, writing, and speaking on these topics since joining Northeastern University as a professor of computer science in 2015. He has a PhD from Sheffield Hallam University, UK, and is a senior member of the IEEE Computer Society.

Colophon

The animal on the cover of *Foundations of Scalable Systems* is a dusky grouper (*Epinephelus marginatus*), also known as the yellowbelly rock cod or yellowbelly grouper. It is common in the Mediterranean Sea, and its range stretches from the Iberian Peninsula along the coast of Africa to Mozambique and from Brazil to northern Argentina. Dusky groupers are normally found in rocky marine areas from the surface down to a depth of about 300 meters. They are ambush feeders, hiding among the rocks and then sucking in prey and swallowing it whole.

Like other groupers, they have large, oval bodies and wide mouths with protruding lower jaws. Dusky groupers have dark reddish-brown or grayish heads with yellow bellies and pale blotches on the head and body. They can reach up to five feet long and can weigh over a hundred pounds. All dusky groupers begin adult life as females and begin to breed at around five years of age, but they develop into males between their ninth and sixteenth years. They live up to 50 years in the wild.

The dusky grouper is a popular food fish, leading it to become a victim of overfishing. Although conservation efforts are being taken, the species is classified as vulnerable. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Johnson's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.